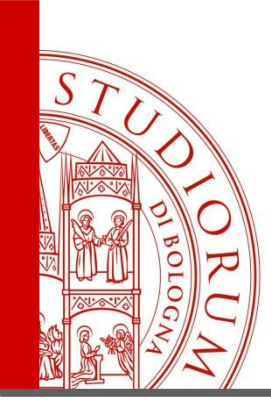


Università degli Studi di Bologna
Corso di Laurea in Ingegneria Informatica

Diagrammi UML

Ingegneria del Software T

Prof. MARCO PATELLA
Dipartimento di Informatica – Scienza e Ingegneria (DISI)

Premessa

- In questo blocco vedremo i principali Diagrammi UML per la fasi di Analisi del Problema e del Progetto
  - per ora non vedremo i diagrammi di deployment e dei componenti
- In particolare:
  - *Diagramma dei Package* e *Diagramma delle Classi*
    → parte "statica" dell'Architettura Logica
    e dell'Architettura del Sistema
  - *Diagramma di Sequenza*
    → parte "interazione" dell'Architettura Logica
    e dell'Architettura del Sistema
  - *Diagramma di Stato* e *Diagramma delle Attività*
    → parte "comportamentale" dell'Architettura Logica
    e dell'Architettura del Sistema

Premessa

- Per ogni Diagramma vedremo solo i concetti fondamentali
- In caso di dubbi fare riferimento alla specifica UML 2.5.1
- Attenzione che nelle lezioni successive si darà per scontata la conoscenza dei diagrammi!!!

Unified Modeling Language

Unified Modeling Language

- È un ***linguaggio*** che serve per visualizzare, specificare, costruire, documentare un sistema e gli elaborati prodotti durante il suo sviluppo
- Ha una semantica e una notazione standard, basate su un ***metamodello*** integrato, che definisce i costrutti forniti dal linguaggio
- La notazione (e la semantica) è **estensibile** e **personalizzabile**
- È utilizzabile per la modellazione durante tutto il ciclo di vita del software (dai requisiti al testing) e per piattaforme e domini diversi

Unified Modeling Language

- Combina approcci di:
  - modellazione dati (Entity/Relationship)
  - business Modeling (workflow)
  - modellazione a oggetti
  - modellazione di componenti
- Prevede una serie di diagrammi standard, che mostrano differenti viste architetturali del modello di un sistema

Unified Modeling Language

- UML è ***un linguaggio e non un processo di sviluppo***
- UML propone un ricco insieme di elementi a livello utente; tuttavia è alquanto informale sul modo di utilizzare al meglio i vari elementi
  - *ciò implica che per* comprendere un diagramma un lettore deve conoscere il contesto in cui esso è collocato

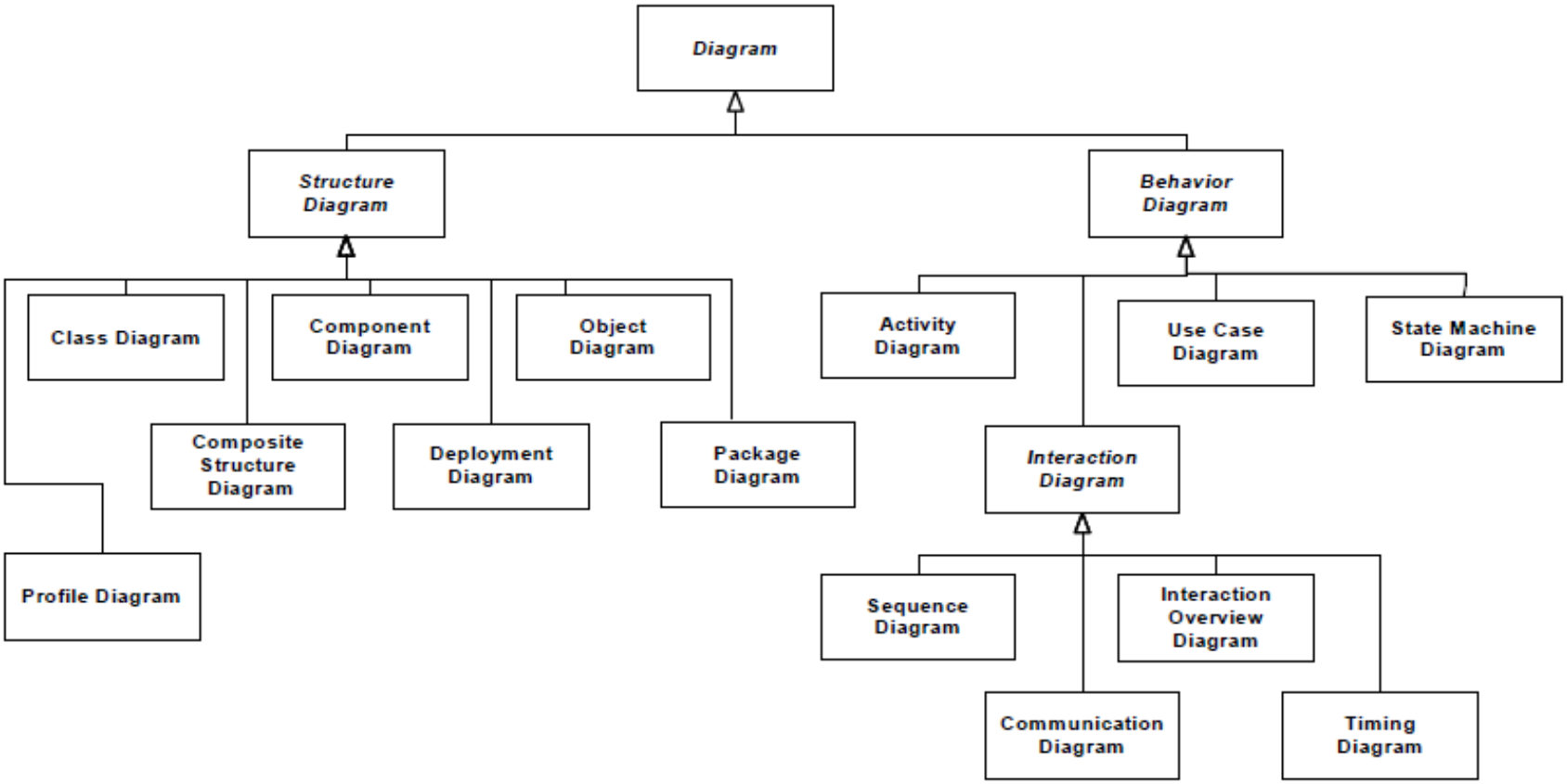
Diagrammi di UML 2.5.1

- Diagrammi di struttura:
  - diagramma delle classi (class)
  - diagramma delle strutture composite (composite structure)
  - diagramma dei componenti (component)
  - diagramma di deployment (deployment)
  - diagramma dei package (package)
  - diagramma dei profili (profile)

Diagrammi di UML 2.5.1

- Diagrammi di comportamento:
  - diagramma dei casi d'uso (use case)
  - diagramma di stato (state machine)
  - diagramma delle attività (activity)
  - **diagrammi di interazione:**
    - diagramma di comunicazione (communication)
    - diagramma dei tempi (timing)
    - diagramma di sintesi delle interazioni (interaction overview)
    - diagramma di sequenza (sequence)

Diagrammi di UML 2.5.1



UML 2.5.1 e Visio

- Visio 2016 gestisce «nativamente» solo alcuni dei diagrammi di UML 2.5.1
  - classi
  - attività
  - sequenza
  - stato
  - casi d'uso
- È possibile però scaricare uno stencil per poter gestire anche gli altri diagrammi

  http://softwarestencils.com/uml/index.html

Diagramma dei Package

Package

- Un package è utilizzato per raggruppare elementi e fornire loro un **namespace**
- Un package può essere innestato in altri package
- NAMESPACE: è una porzione del modello nella quale possono essere definiti e usati dei nomi
- In un namespace ogni nome ha un significato univoco

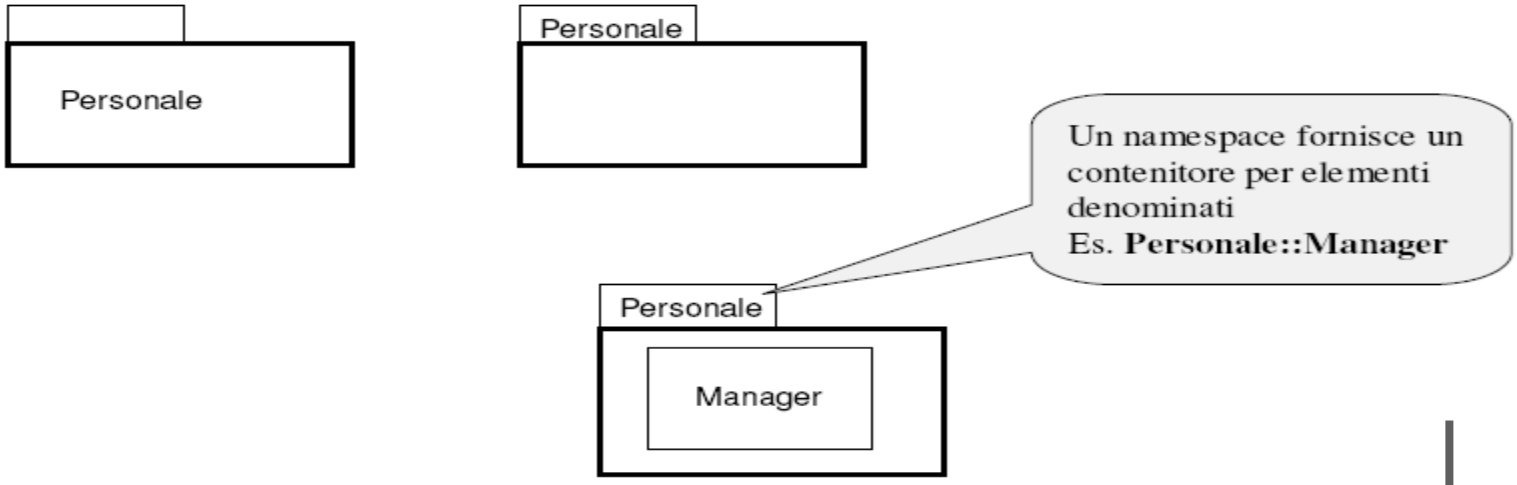
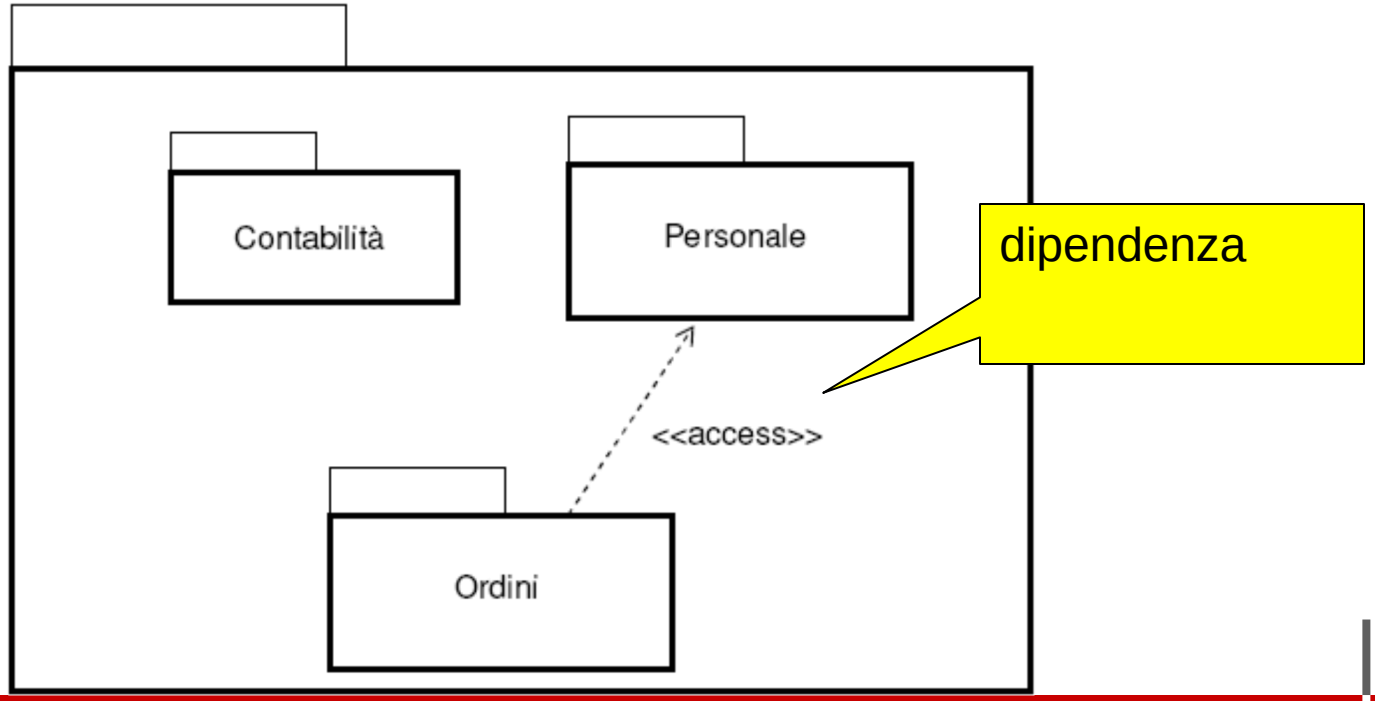


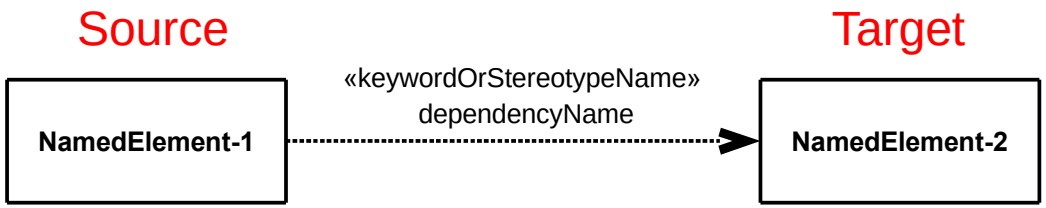
Diagramma dei Package

- Un diagramma dei package è un diagramma che illustra come gli elementi di modellazione sono organizzati in package e le relazioni (dipendenze) tra i package



Dipendenze

- UML permette di rappresentare relazioni che NON sussistono fra istanze nel dominio rappresentato, ma sussistono fra gli elementi del modello UML stesso o fra le astrazioni che tali elementi rappresentano
- **Dipendenza:**
  - è rappresentata da una linea tratteggiata orientata che va da un elemento dipendente (Source) ad uno indipendente (Target)



Dipendenze

- Una dipendenza indica che cambiamenti dell'elemento **indipendente** influenzano l'elemento **dipendente**
  - modificano il significato dell'elemento dipendente
  - causano la necessità di modificare anche l'elemento dipendente perché i significati sono dipendenti
- UML mette a disposizione nove diversi tipi di dipendenza, ma per i nostri fini consideriamo quasi sempre <<use>>

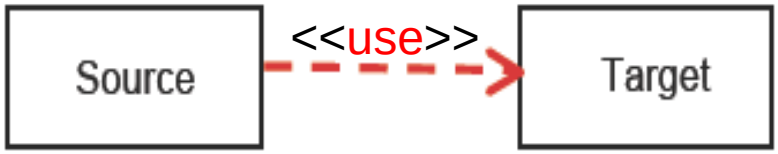
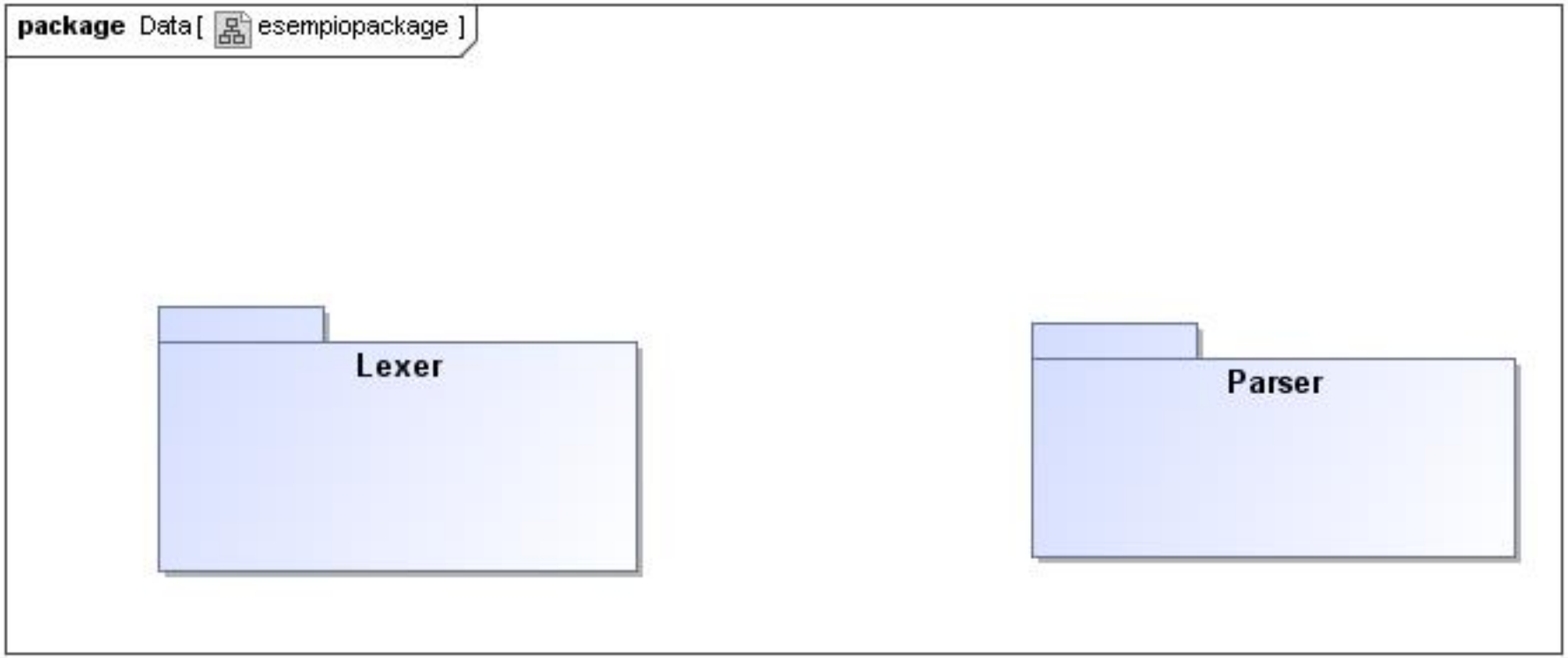


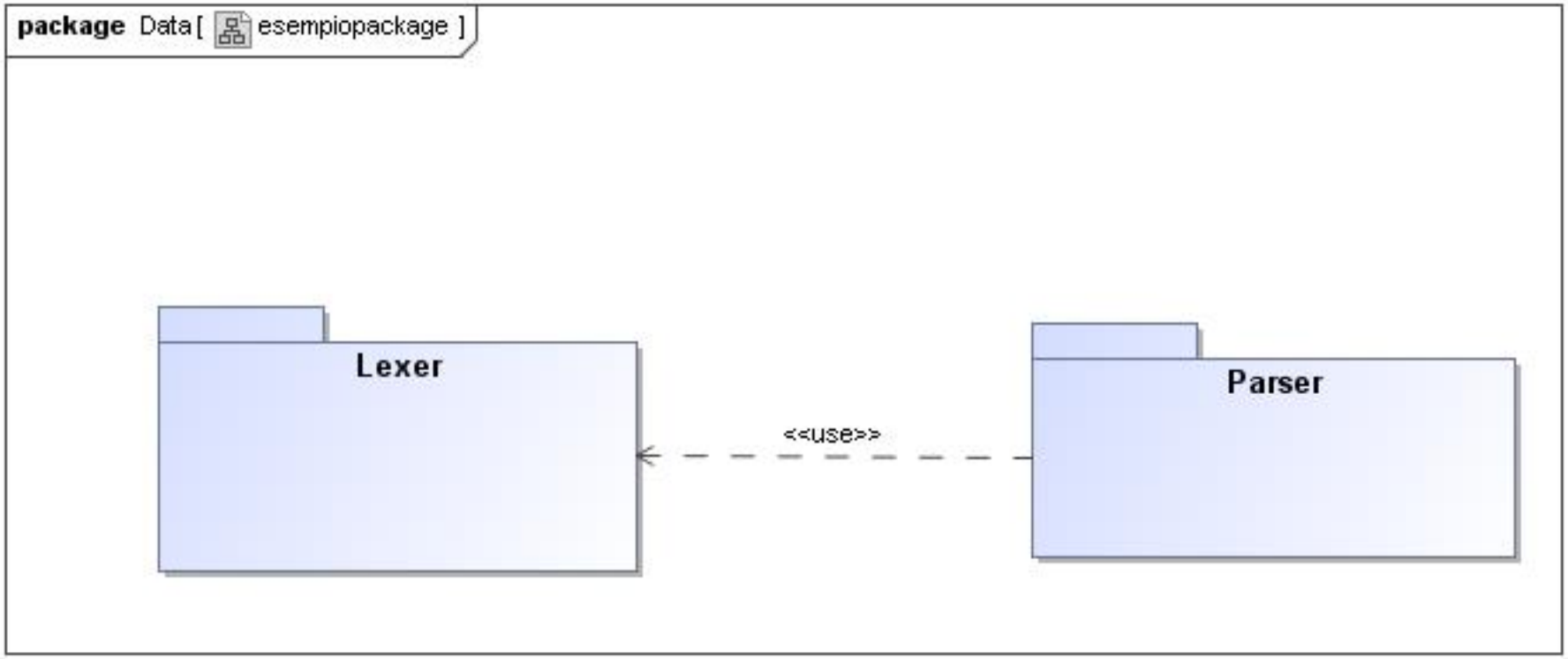
Diagramma dei Package

- Quando si usa il diagramma dei package
  per definire la parte strutturale dell'architettura logica
  ricordare sempre che si stanno esprimendo ***dipendenze logiche*** che sussistono tra le entità del problema
- Non è detto che tali dipendenze rimangano tali
  anche nella fase di progettazione
- Esempio: in sistema per l'interpretazione
  di una grammatica si hanno due parti fondamentali
  - Lexer → strumento che legge e spezza
    una sequenza di caratteri in sotto-sequenze dette "token"
  - Parser prende in ingresso i token generati, li analizza e li elabora
    in base ai costrutti specificati nella grammatica stessa

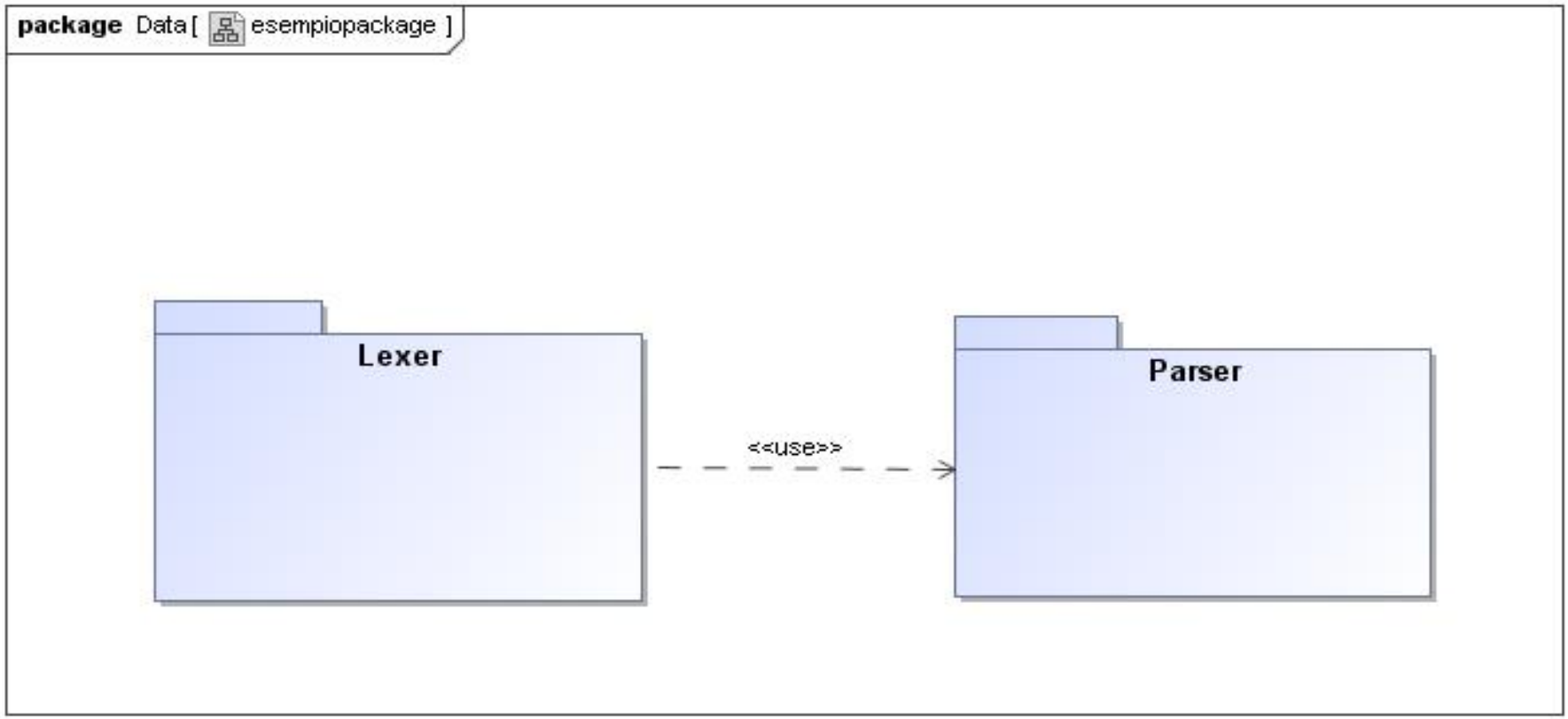
Esempio



Analisi del Problema



Progetto



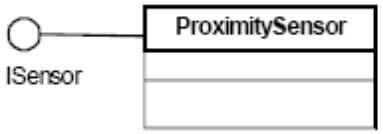
Interfacce

Interfaccia

- Le interfacce forniscono un modo per partizionare e caratterizzare gruppi di proprietà
- Un'interfaccia non deve specificare come possa essere implementata, ma semplicemente quello che è necessario per poterla realizzare
- Le entità che realizzano l'interfaccia dovranno fornire una "***vista pubblica***" (attributi, operazioni, comportamento osservabile all'esterno) conforme all'interfaccia stessa
- Se un'interfaccia dichiara un attributo, non significa necessariamente che l'elemento che realizza l'interfaccia debba avere quell'attributo nella sua implementazione, ma solamente che esso apparirà così a un osservatore esterno

Interfaccia: notazione

Interfaccia fornita



Interfaccia richiesta

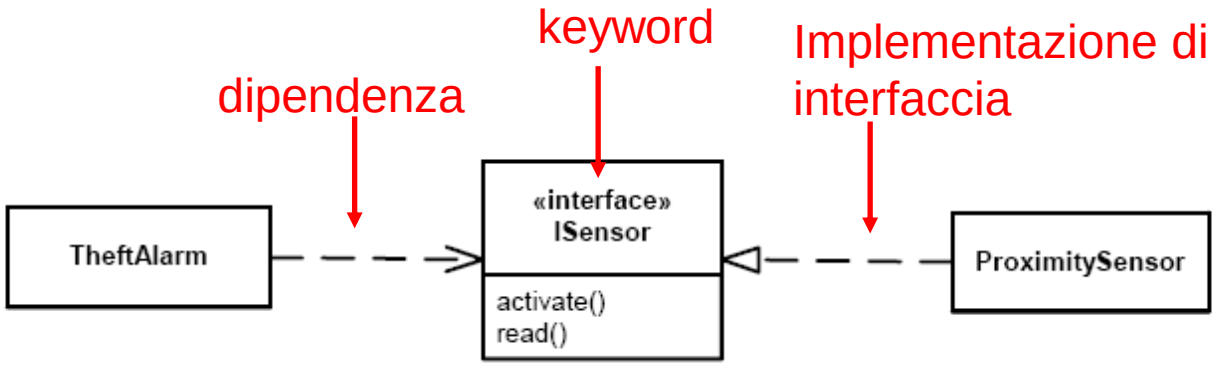
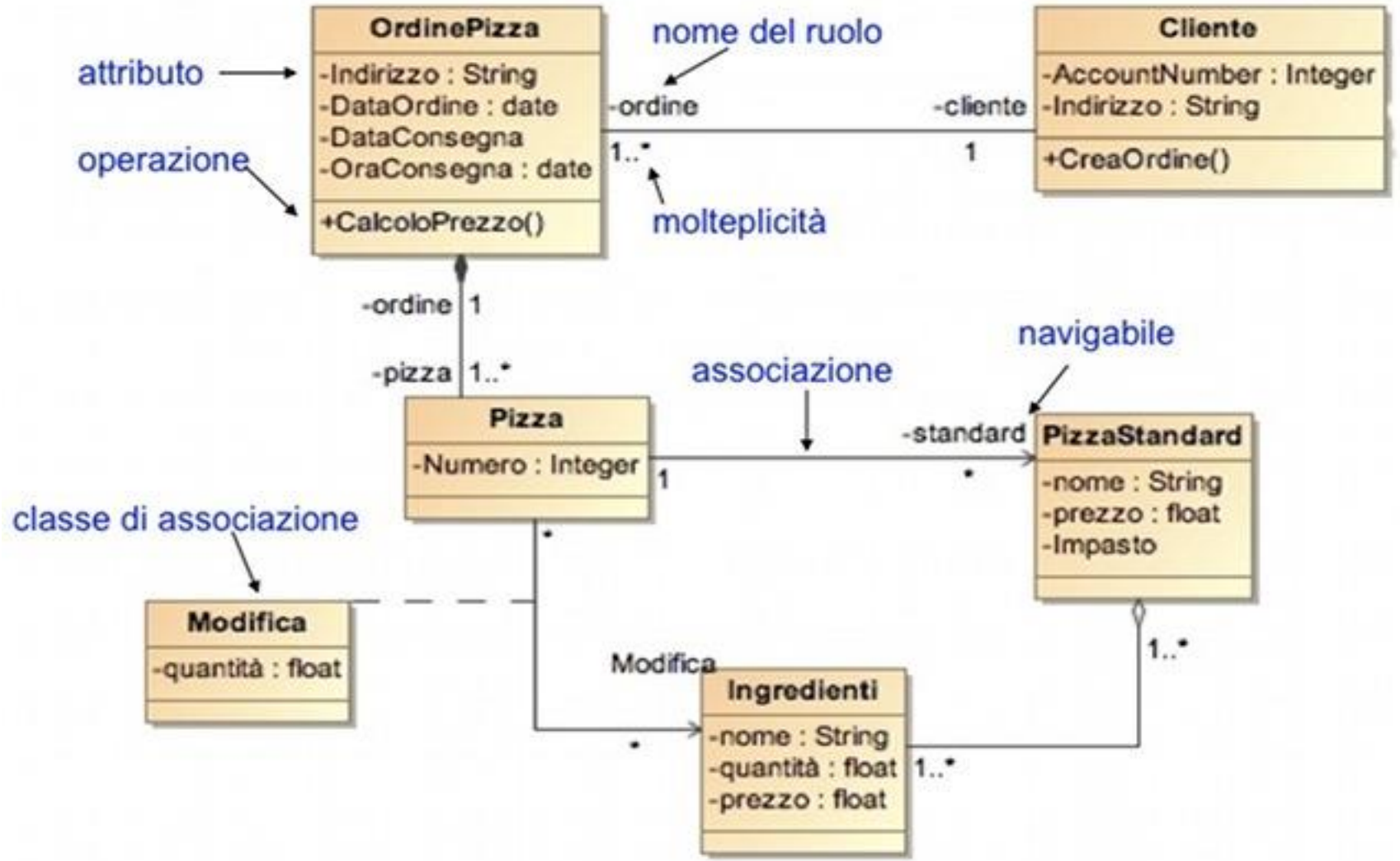


Diagramma delle Classi

Diagramma delle Classi

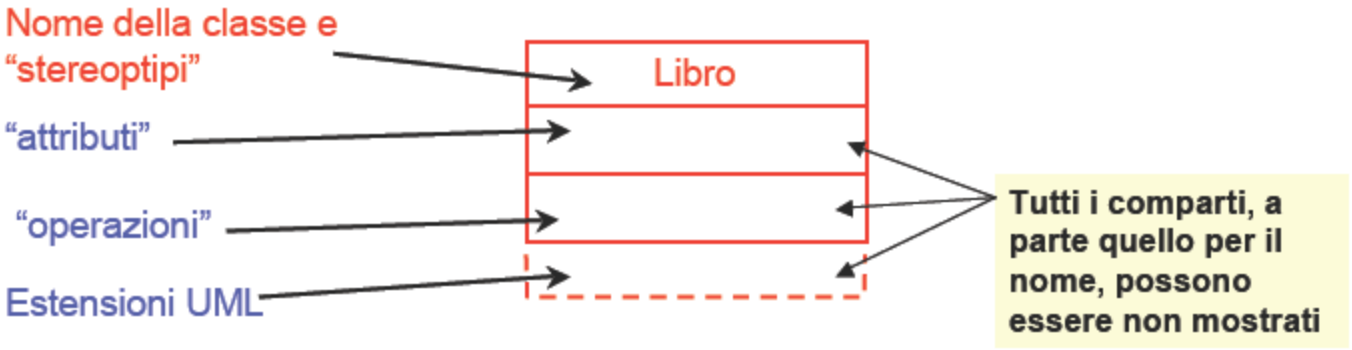
- Un **diagramma delle classi** descrive il tipo degli oggetti facenti parte di un sistema e le varie tipologie di relazioni statiche tra di essi
- I diagrammi delle classi mostrano anche le *proprietà* e le *operazioni* di una classe e i *vincoli* che si applicano alla classe e alle relazioni tra classi
- Le ***proprietà*** rappresentano le caratteristiche strutturali di una classe:
  - sono un unico concetto, rappresentato però con due notazioni molto diverse: ***attributi*** e ***associazioni***
  - benché il loro aspetto grafico sia molto differente, concettualmente sono la stessa cosa

Esempio


OrdinePizza
-Indirizzo : String
-DataOrdine : date
-DataConsegna
-OraConsegna : date
+CalcoloPrezzo()
attributo
operazione
nome del ruolo
-ordine
-cliente
1..*
1
molteplicità
Cliente
-AccountNumber : Integer
-Indirizzo : String
+CreaOrdine()
-ordine 1
-pizza 1..*
Pizza
-Numero : Integer
associazione
navigabile
-standard
1
*
PizzaStandard
-nome : String
-prezzo : float
-Impasto
classe di associazione
Modifica
-quantità : float
Modifica
Ingredienti
-nome : String
-quantità : float
-prezzo : float
1..*
1..*

Classe

- Una ***classe*** modella un insieme di entità (le istanze della classe) aventi tutti lo stesso tipo di caratteristiche (attributi, associazioni, operazioni...).
- Ogni classe è descritta da:
  - un **nome**
  - un insieme di **caratteristiche (feature)**: attributi, operazioni, ...



Attributi

- La notazione degli **attributi** descrive una proprietà con una riga di testo all'interno del box della classe
- La forma completa è:
  - **visibilità nome:tipo molteplicità=default {stringa di proprietà}**
- Un esempio di attributo è:
  - stringa: String [10] = "Pippo" {readOnly}
- L'unico elemento necessario è il nome
  - Visibilità: attributo pubblico (+), privato (-) o protected (#)
  - Nome: corrisponde al nome dell'attributo
  - Tipo: vincolo sugli oggetti che possono rappresentare l'attributo
  - Default: valore dell'attributo in un oggetto appena creato
  - Stringa di proprietà: caratteristiche aggiuntive (readOnly)
  - Molteplicità: …

Molteplicità

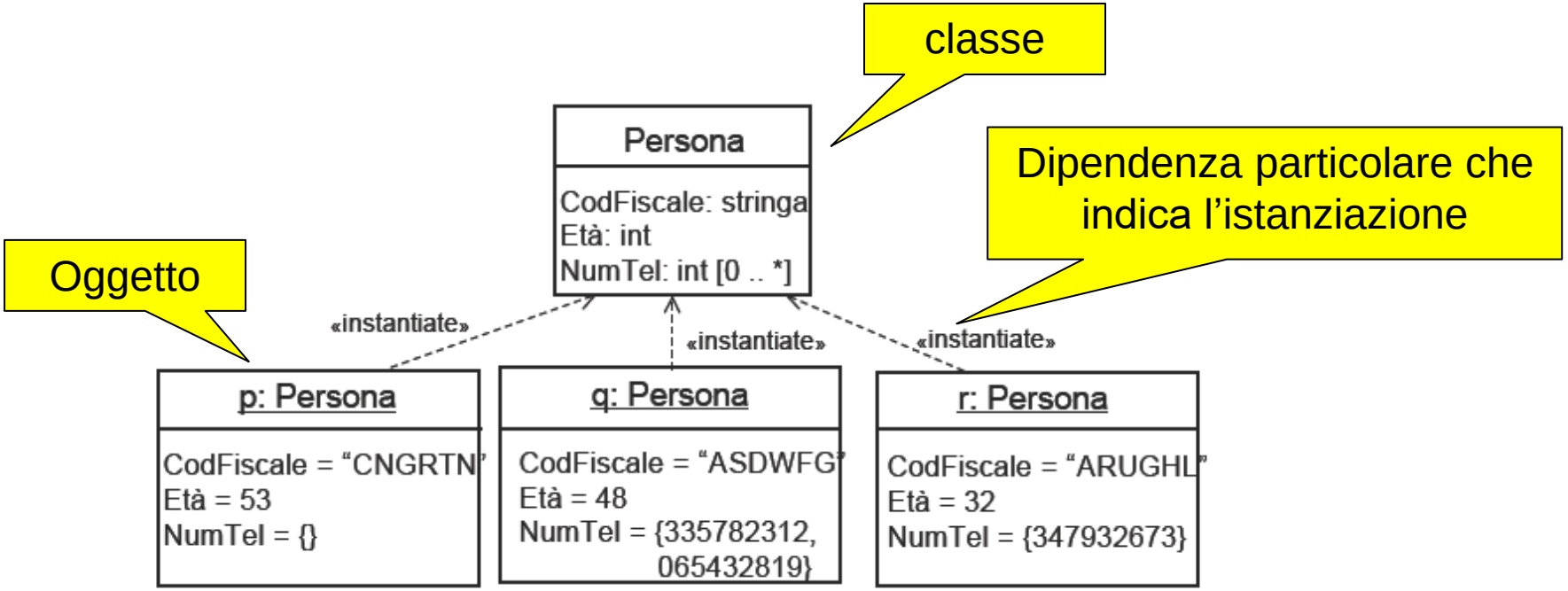
- È l'indicazione di quanti oggetti possono entrare a far parte di una proprietà
- Le molteplicità più comuni sono:
  - **1, 0..1, ***
- In modo più generale, le molteplicità si possono definire indicando gli estremi inferiore e superiore di un intervallo (per esempio ***2..4***).
- Molti termini si riferiscono alla molteplicità degli attributi:
  - **Opzionale:** indica un limite inferiore di 0
  - **Obbligatorio:** implica un limite inferiore di 1 o più
  - **A un solo valore:** implica un limite superiore di 1
  - **A più valori:** implica che il limite superiore sia maggiore di 1, solitamente *

Visibilità

- È possibile etichettare ogni operazione o attributo con un identificatore di visibilità
- UML fornisce comunque quattro abbreviazioni per indicare la visibilità:
  - + (public)
  - - (private)
  - ~ (package)
  - # (protected)

Attributi: Molteplicità

- Nelle istanze, il **valore di un attributo multi-valore** si indica mediante un insieme



Operazioni

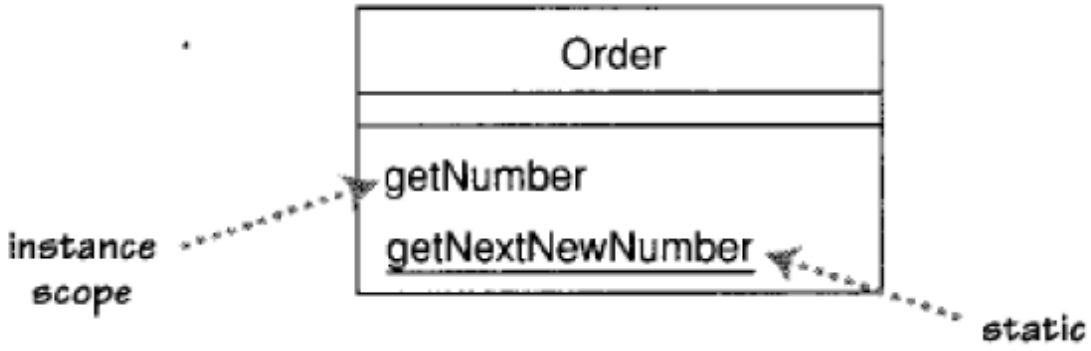
- Le operazioni sono le azioni che la classe sa eseguire, e in genere si fanno corrispondere direttamente ai metodi della corrispondente classe a livello implementativo
- Le operazioni che manipolano le proprietà della classe di solito si possono dedurre, per cui non sono incluse nel diagramma
- La sintassi UML completa delle operazioni è

  visibilità nome (lista parametri) : tipo ritorno {stringa di propr}

  - Visibilità: operazione pubblica (+) o privata (-)
  - Nome: stringa
  - Lista parametri: lista parametri dell'operazione
  - Tipo di ritorno: tipo di valore restituito dall'operazione, se esiste
  - Stringa di proprietà: caratteristiche aggiuntive che si applicano all'operazione

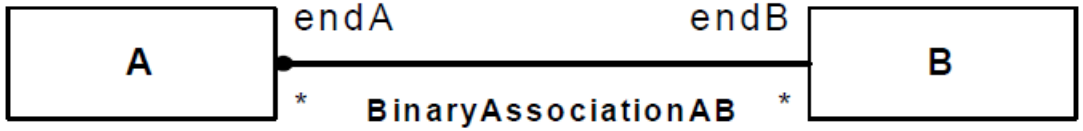
Operazioni e Attributi Statici

- UML chiama **static** un'operazione o un attributo che si applicano a una classe anziché alle sue istanze
- Questa definizione equivale a quella dei membri statici nei linguaggi come per esempio java e C#
- Le caratteristiche statiche vanno sottolineate sul diagramma



Associazioni

- Le associazioni sono un altro modo di rappresentare le proprietà
- Gran parte dell'informazione che può essere associata a un attributo si applica anche alle associazioni
- Un'associazione è una linea continua che collega due classi, orientata dalla classe sorgente a quella destinazione
- Il nome e la molteplicità vanno indicati vicino all'estremità finale dell'associazione:
  - la classe destinazione corrisponde al tipo della proprietà



Associazioni

- Assegnare dei nomi ai "**ruoli**" svolti da ciascun elemento di un associazione
- Anche nei casi in cui non è strettamente necessario, il ruolo può essere utile per aumentare la leggibilità del diagramma

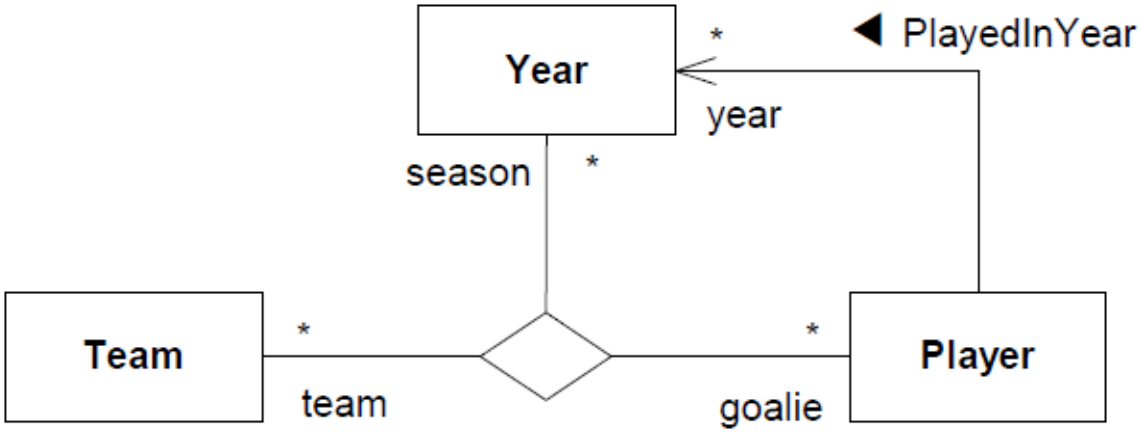
Associazioni Bidirezionali

- Una tipologia di associazione è quella **bidirezionale (o binaria)**, costituita da una coppia di proprietà collegate, delle quali una è l'inversa dell'altra
- Il collegamento inverso implica che, se seguite il valore di una proprietà e poi il valore della proprietà collegata, dovreste ritornare all'interno di un insieme che contiene il vostro punto di partenza

la natura bidirezionale dell'associazione è palesata dalle **frecce di navigabilità** aggiunte a entrambi i capi della linea

Associazioni Ternarie

- Quando si hanno associazioni ternarie (o che coinvolgono più classi) si introduce il simbolo "diamante"



Associazioni: molteplicità

- La specifica UML (vista fino a ora) dichiara che la molteplicità di un'associazione è

the multiplicity of instances of that entity (the range of number of objects that participate in the association from the perspective of the other end)

- Tale definizione (derivata dalla specifica E/R originale) non può però applicarsi alle associazioni multiple…

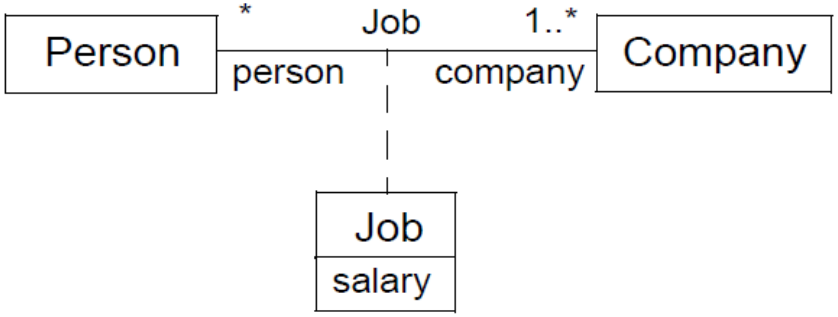
Associazioni: molteplicità

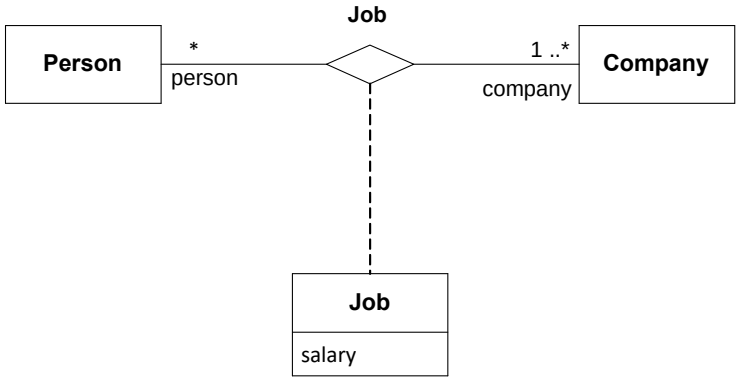
- Pertanto, come già visto nel corso SIT, la notazione usata in questo corso (e in altri) prevede che la molteplicità di un'associazione rappresenti:

il numero (minimo e massimo)
di istanze dell'associazione a cui un'istanza dell'entità può partecipare

Classi di Associazione

- Le classi di associazione permettono di aggiungere attributi, operazioni e altre caratteristiche che sono proprie dell'associazione

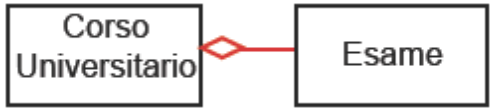




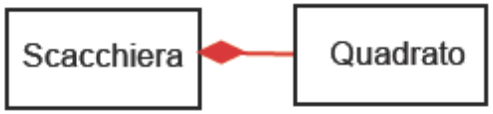
- La classe di associazione aggiunge implicitamente un vincolo extra: ci può essere solo un'istanza della classe di associazione tra ogni coppia di oggetti associati

Aggregazione e Composizione

- **Aggregazione**:
  - è un associazione che corrisponde a una relazione intuitiva Intero-Parte ("**part-of**")
  - è rappresentata da un **diamante vuoto** sull'associazione, posto vicino alla classe le cui istanze sono gli "interi"

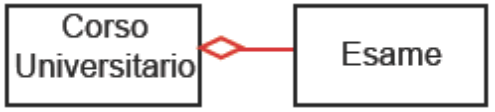


- **Composizione:**
  - è un aggregazione che rispetta due vincoli ulteriori:
    - una parte può essere inclusa in al massimo un intero in ogni istante
    - solo l'oggetto intero può creare e distruggere le sue parti
  - è rappresentata da un **diamante pieno** vicino alla classe che rappresenta gli "interi"

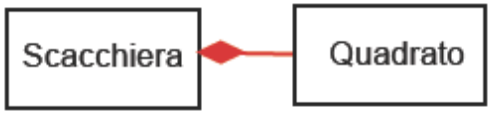


Aggregazione e Composizione

- **Aggregazione**:
  - è una relazione **binaria**
  - può essere **ricorsiva**

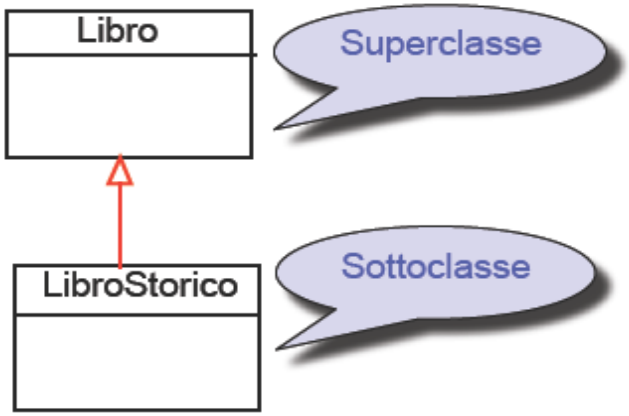


- **Composizione**:
  - se l'oggetto che compone viene distrutto, anche i figli vengono distrutti, …
  - … anche se i figli possono essere creati/distrutti in momenti diversi dalla creazione/distruzione dell'oggetto che compone
  - può essere **ricorsiva**



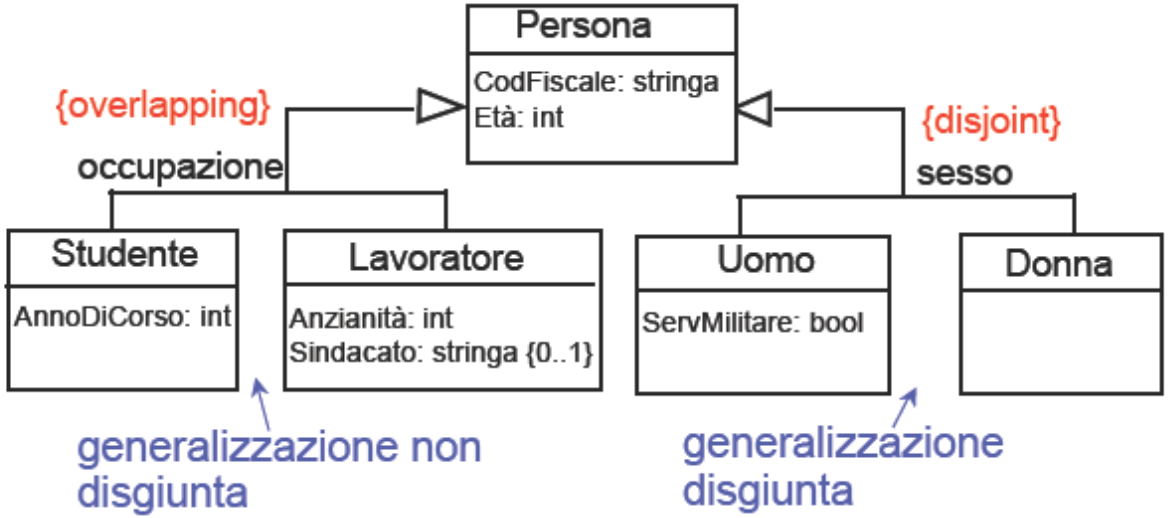
Generalizzazione

- La generalizzazione è indicata con una freccia vuota fra due classi dette **sottoclasse** e **superclasse**
- Il significato della generalizzazione è il seguente: ogni istanza della sottoclasse è anche istanza della superclasse



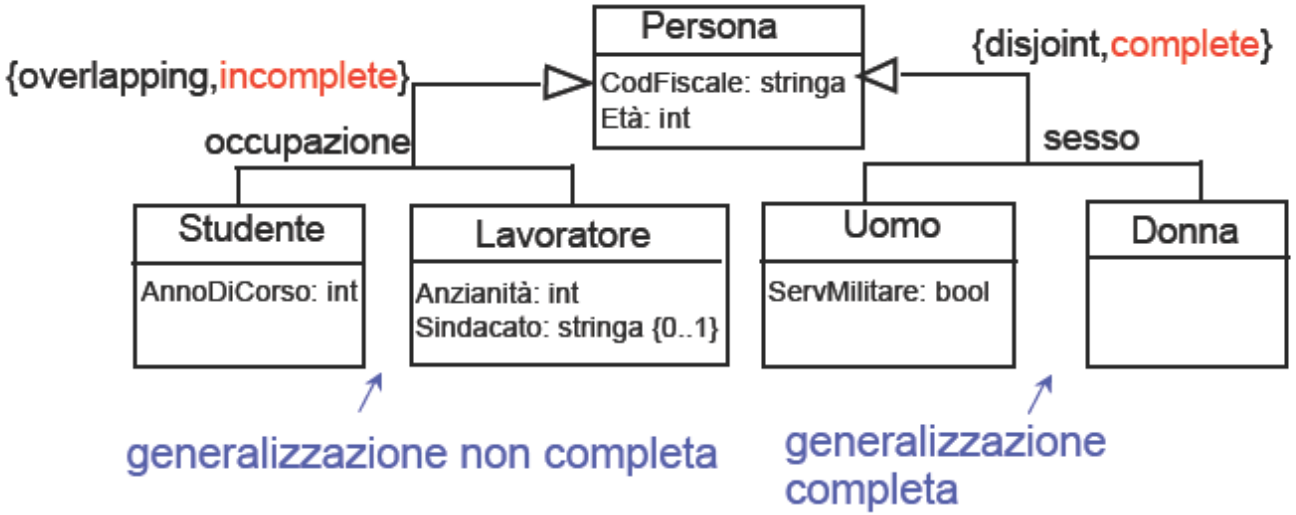
Generalizzazione

- La stessa superclasse può partecipare a diverse generalizzazioni
- Una generalizzazione può essere **disgiunta**, cioè le sottoclassi sono disgiunte (non hanno istanze in comune), o no



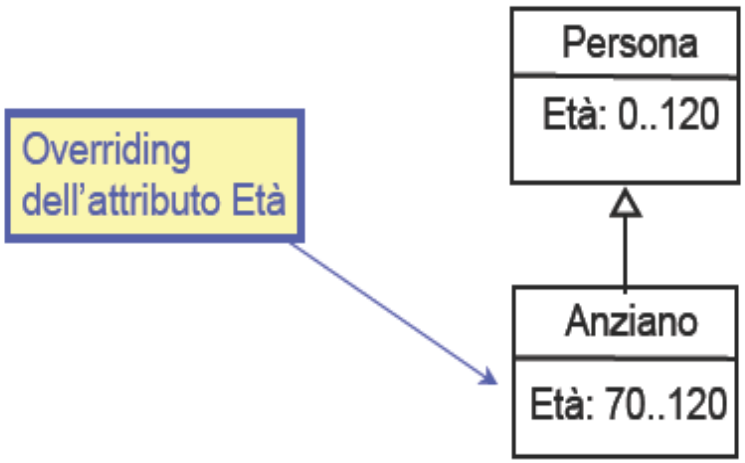
Generalizzazione

- Una generalizzazione può essere **completa** (l'unione delle istanze delle sottoclassi è uguale all'insieme delle istanze della superclasse) o no
- **Attenzione: i valori di default** sono {incomplete, disjoint}



Generalizzazione

- In una generalizzazione la sottoclasse non solo può avere caratteristiche aggiuntive rispetto alla superclasse, ma può anche **sovrascrivere** (**overriding**) le proprietà ereditate dalla superclasse



In UML 2.0 il termine "overriding" è sostituito da "ridefinizione"(redefining).

Eventuali vincoli fra proprietà della superclasse e proprietà corrispondente nella sottoclasse dipendono dal significato che le proprietà assumono nel particolare contesto di applicazione.

Ad es. spesso si assume questo vincolo: "un attributo può essere ridefinito solo sostituendo il tipo con un sottotipo".

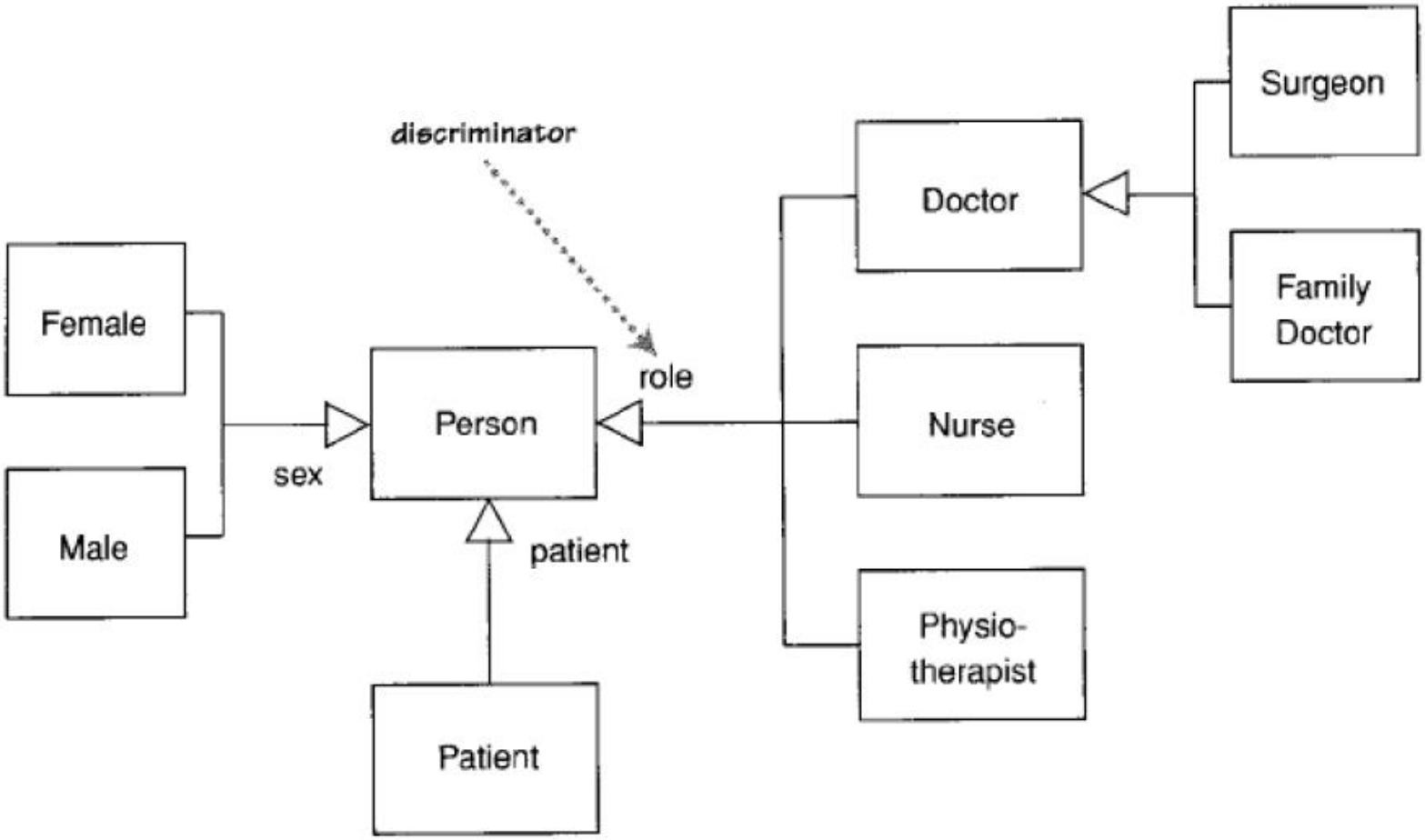
Generalizzazione Multipla

- Con la **generalizzazione singola** un oggetto appartiene a un solo tipo, che può eventualmente ereditare dai suoi tipi padre
- Con la ***generalizzazione multipla*** un oggetto può essere descritto da più tipi, non necessariamente collegati dall'ereditarietà
- Si noti che la generalizzazione multipla è una cosa ben diversa dall'ereditarietà multipla
  - *Ereditarietà multipla*: un tipo può avere più supertipi, ma ogni oggetto deve sempre appartenere a un suo tipo ben definito
  - *Generalizzazione multipla*: un oggetto viene associato a più tipi senza che per questo debba esserne appositamente definito un altro

Generalizzazione Multipla

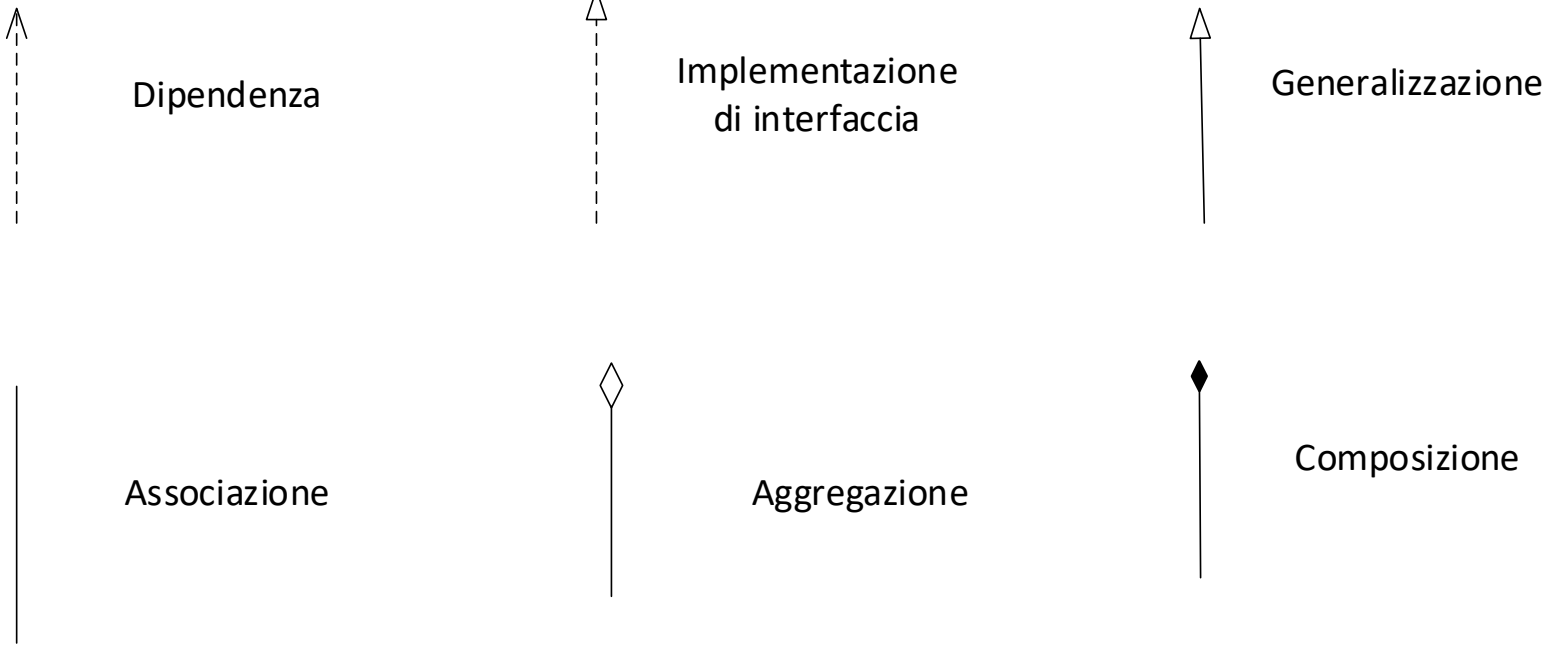
- Se si usa la generalizzazione multipla, ci si deve assicurare di rendere chiare le combinazioni “legali”
- *Per questo fatto, UML pone ogni relazione di generalizzazione in un* ***insieme di generalizzazione***
- Sul diagramma delle classi, la freccia che indica una generalizzazione va etichettata con il nome del rispettivo insieme
- La generalizzazione singola corrisponde all’uso di un singolo anonimo insieme di generalizzazione
- Gli insiemi di generalizzazione sono disgiunti per definizione
  - Ogni istanza del supertipo può essere istanza di uno dei sottotipi all’interno di quel sottoinsieme

Generalizzazione Multipla



Relazioni tra classi: sintassi

- Attenzione all'uso corretto delle frecce
- UML è un linguaggio (anche se grafico) e scambiare una freccia per un'altra è un errore non da poco



Classi Astratte

- Una ***classe astratta*** è una classe che non può essere direttamente istanziata: per farlo bisogna prima crearne una sottoclasse concreta
- Tipicamente, una classe astratta ha una o più operazioni astratte
- Un'operazione astratta non ha implementazione
  - è costituita dalla sola dichiarazione, resa pubblica affinché le classi client possano usufruirne
- Il modo più diffuso di indicare una classe o un'operazione astratta in UML è scriverne il nome in *corsivo*
- Si possono anche rendere astratte le proprietà indicandole direttamente come tali o rendendo astratti i metodi d'accesso

Classi Astratte

- A cosa servono?
  - servono come superclassi comuni
    per un insieme di sottoclassi concrete
  - queste sottoclassi, in virtù del subtyping,
    sono in qualche misura compatibili e intercambiabili
    fra di loro
  - infatti sono tutte sostituibili con la superclasse
    - sulle istanze di ognuna di esse possiamo invocare i metodi ereditati dalla classe astratta

Enumerazioni

- Le enumerazioni sono usate per mostrare un insieme di valori prefissati che non hanno altre proprietà oltre al loro valore simbolico
- Sono rappresentate con una classe marcata dalla parola chiave «enumeration»

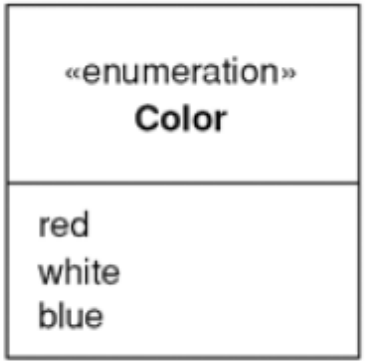
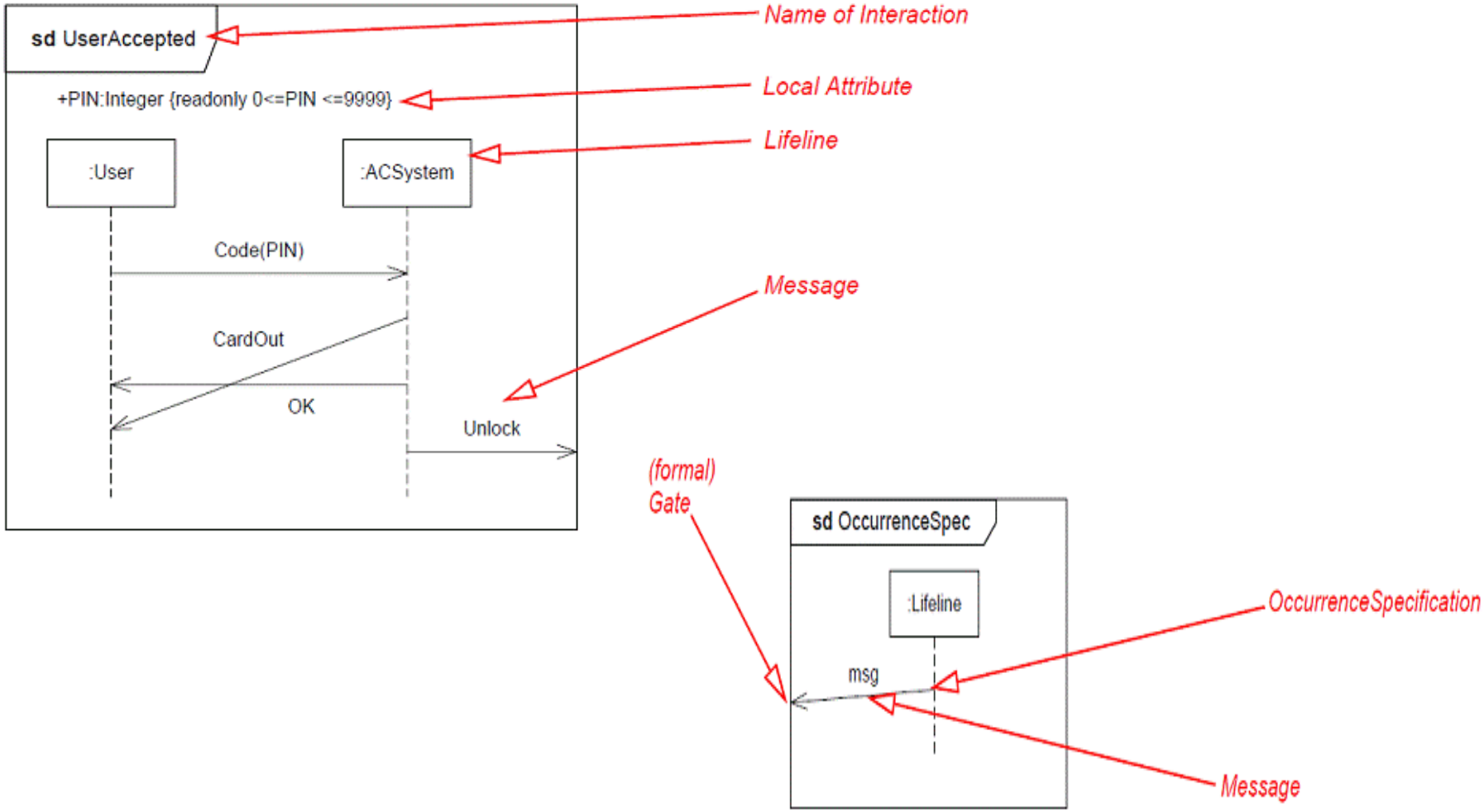


Diagramma di Sequenza

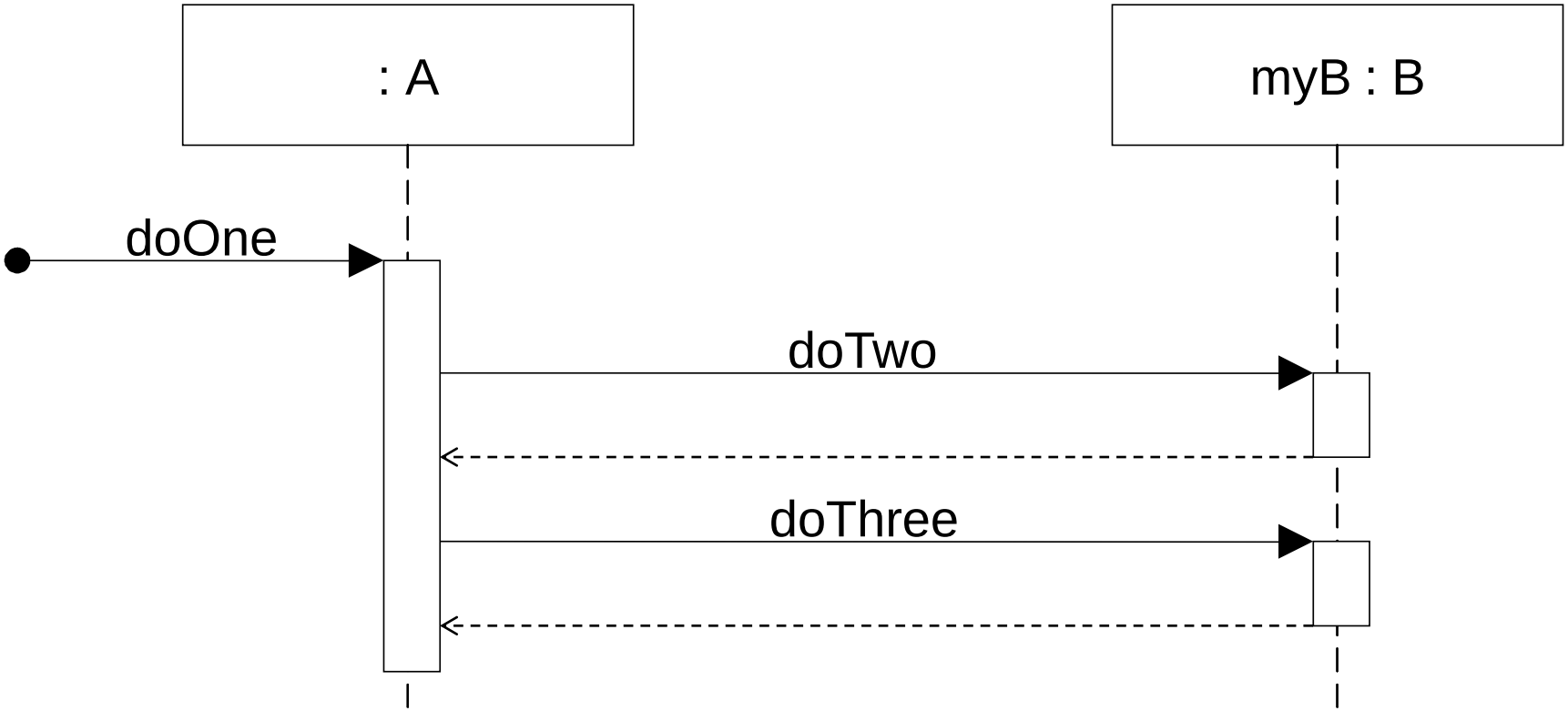
Diagramma di Sequenza

- Diagramma che illustra le interazioni tra le classi / entità disponendole lungo una sequenza temporale
- In particolare mostra i soggetti (chiamati tecnicamente ***Lifeline***) che partecipano all'interazione e la sequenza dei messaggi scambiati
- In ascissa troviamo i diversi soggetti (non necessariamente in ordine di esecuzione), mentre in ordinata abbiamo la scala dei tempi sviluppata verso il basso

Diagramma di Sequenza



Esempio



Lifeline

- In un diagramma di sequenza, i partecipanti solitamente sono istanze di classi UML caratterizzate da un nome
- La loro vita è rappresentata da una *Lifeline*, cioè una linea tratteggiata verticale ed etichettata, in modo che sia possibile comprendere a quale componente del sistema si riferisce
- In alcuni casi il partecipante non è un'entità semplice, ma composta
  - è possibile modellare la comunicazione fra più sottosistemi, assegnando una Lifeline ad ognuno di essi

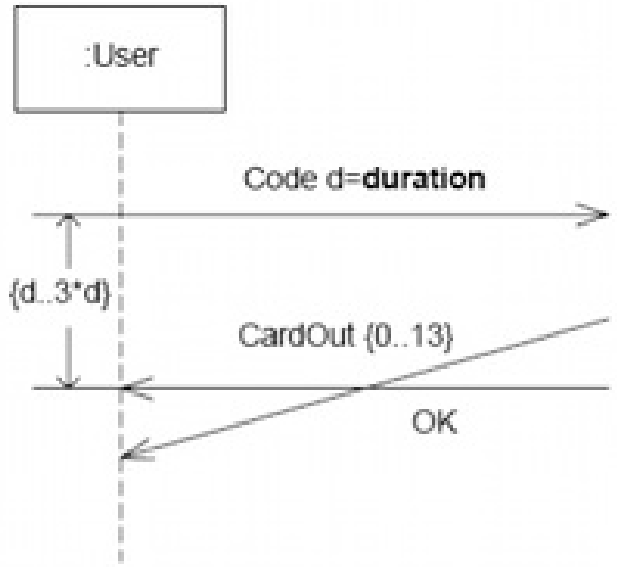
Lifeline

- L'ordine in cui le OccurrenceSpecification (cioè l'invio e la ricezione di eventi) avvengono lungo la Lifeline rappresenta esattamente l'ordine in cui tali eventi si devono verificare
- La distanza (in termini grafici) tra due eventi non ha rilevanza dal punto di vista semantico
- Dal punto di vista notazionale, una Lifeline è rappresentata da un rettangolo che costituisce la "testa" seguito da una linea verticale che rappresenta il tempo di vita del partecipante
- È interessante notare che nella sezione della notazione, viene indicato espressamente che il "rettangolino" che viene apposto sulla Lifeline rappresenta l'attivazione di un metodo

Vincoli Temporali

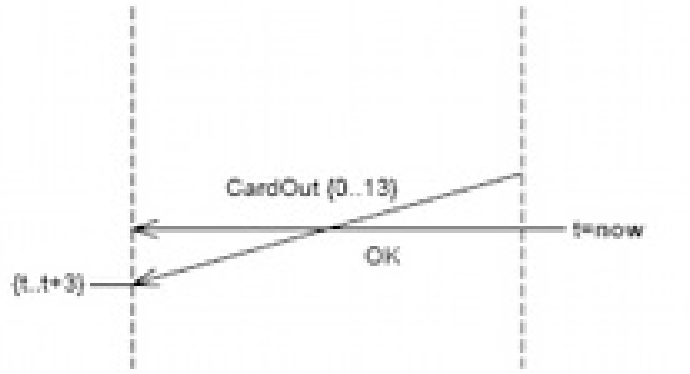
- Per modellare sistemi real-time, o comunque qualsiasi altra tipologia di sistema in cui la temporizzazione è critica, è necessario specificare un istante in cui un messaggio deve essere inviato, oppure quanto tempo deve intercorrere fra un'interazione e un'altra
- Grazie, rispettivamente, a *Time Constraint e Duration Constraint è possibile definire questo genere di* vincoli

Vincoli Temporali

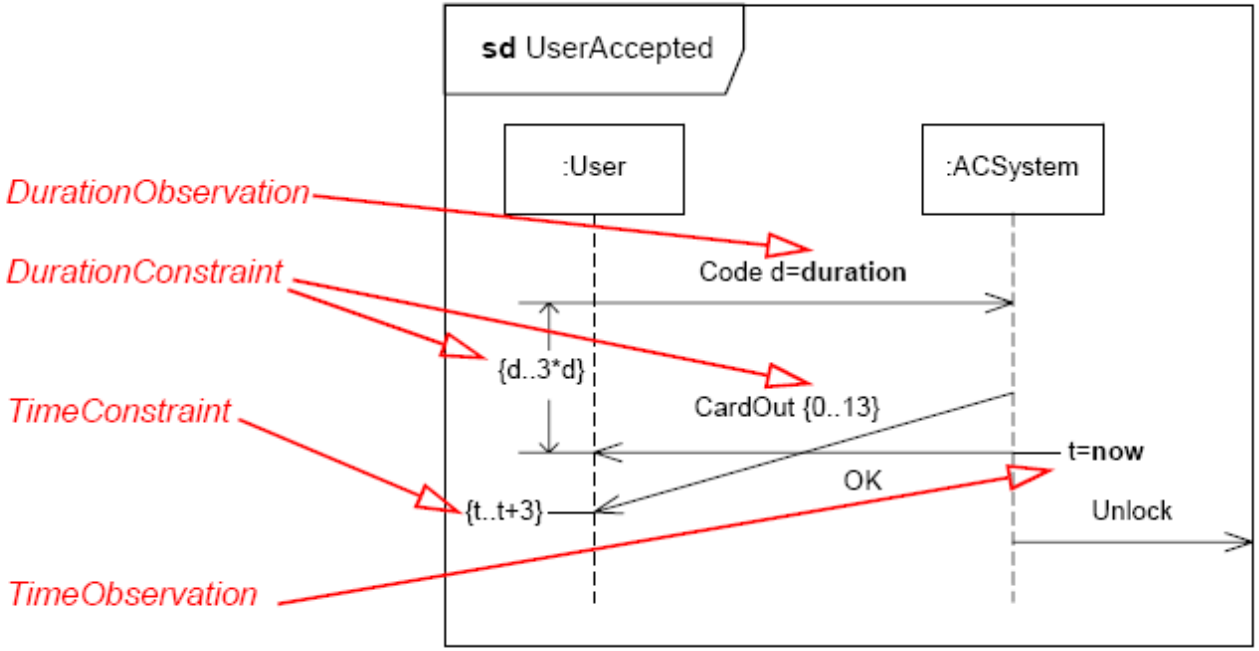


DurationConstraint

TimeConstraint



Vincoli Temporali



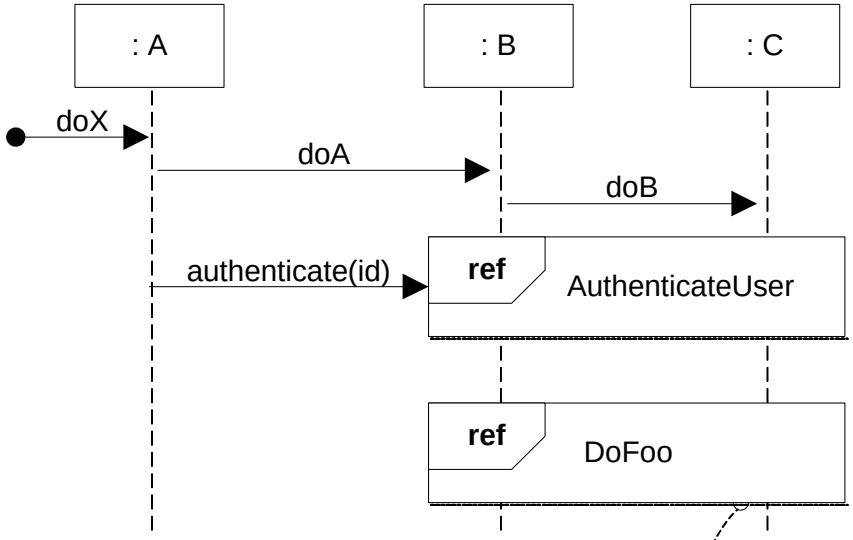
Riferimento ad altri Diagrammi

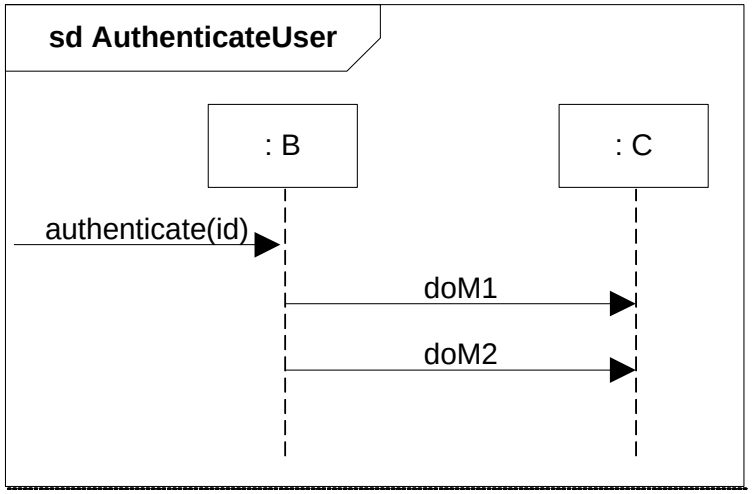
- Spesso i diagrammi di sequenza possono assumere una certa complessità
  - necessità di poter definire comportamenti più articolati come composizione di nuclei di interazione più semplici
- Oppure, se una sequenza di eventi ricorre spesso, potrebbe essere utile definirla una volta e richiamarla dove necessario
- Per questa ragione, UML permette di inserire **riferimenti ad altri diagrammi** e passare loro degli argomenti

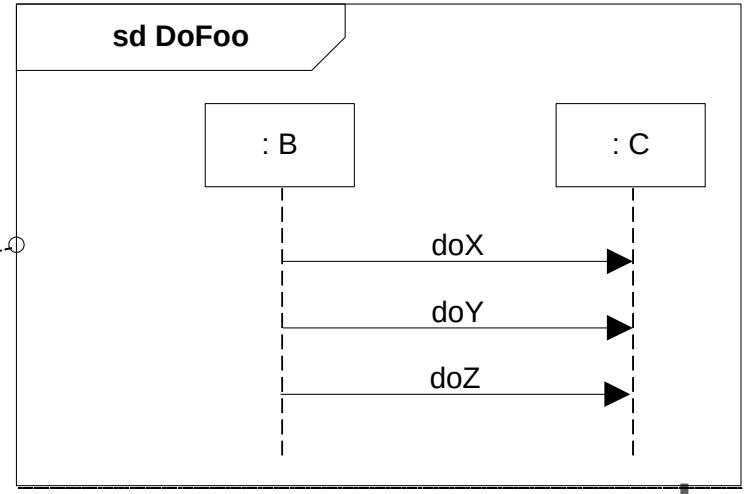
Riferimento ad altri Diagrammi

- Ovviamente ha senso sfruttare quest'ultima opzione solo se il diagramma accetta dei parametri sui quali calibrare l'evoluzione del sistema
- Questi riferimenti prendono il nome di *InteractionUse*
- I punti di connessione tra i due diagrammi prendono il nome di *Gate*
- Un Gate rappresenta un punto di interconnessione che mette in relazione un messaggio al di fuori del frammento di interazione con uno all'interno del frammento

Riferimento ad altri Diagrammi







interaction occurrence

note it covers a set of lifelines

note that the sd frame it relates to has the same lifelines: B and C

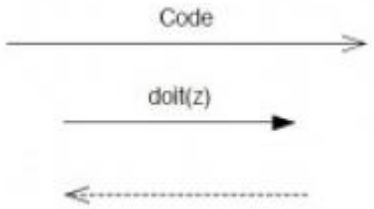
Messaggio

- Un messaggio rappresenta un'interazione realizzata come comunicazione fra Lifeline
- Questa interazione può consistere nella creazione o distruzione di un'istanza, nell'invocazione di un'operazione, o nella emissione di un segnale
- UML permette di rappresentare tipi differenti di messaggi

Tipi di Messaggio

- Se sono specificati mittente e destinatario
  è un ***complete message***
  - *la semantica è rappresentata* quindi dall'occorrenza della coppia di eventi <sendEvent, receiveEvent>
- Se il destinatario non è stato specificato
  è un ***lost message***
  - *in questo caso è noto* solo l'evento di invio del messaggio
- Se il mittente non è stato specificato
  è un ***found message***
  - *in questo caso è noto solo* l'evento di ricezione del messaggio
- Nel caso non sia noto né il destinatario né il mittente
  è un ***unknown message***

Messaggio

Node Type	*Notation*	*Reference*
Message		Messages come in different variants depending on what kind of Message they convey. Here we show an asynchronous message, a call and a reply. These are all *complete* messages.
Lost Message		Lost messages are messages with known sender, but the reception of the message does not happen.
Found Message		Found messages are messages with known receiver, but the sending of the message is not described within the specification.
GeneralOrdering		

Tipi di Messaggio

- Attenzione alle frecce che usate nei messaggi:
  hanno significati diversi:
  - **riga continua freccia piena:** indica un messaggio (**call**) **sincrono** in cui il mittente **aspetta** il completamento dell'esecuzione del destinatario prima di continuare la sua esecuzione.
    - ***Necessita di un messaggio di ritorno*** per sbloccare l'esecuzione del mittente
  - **riga continua freccia vuota:** indica un messaggio **asincrono** in cui il mittente **non aspetta** il completamento dell'esecuzione del destinatario ma continua la sua esecuzione
    - Il valore di ritorno potrebbe o meno essere necessario, dipende dalla semantica
  - **riga tratteggiata freccia vuota:** indica il ritorno di un messaggio

Diagramma di Sequenza

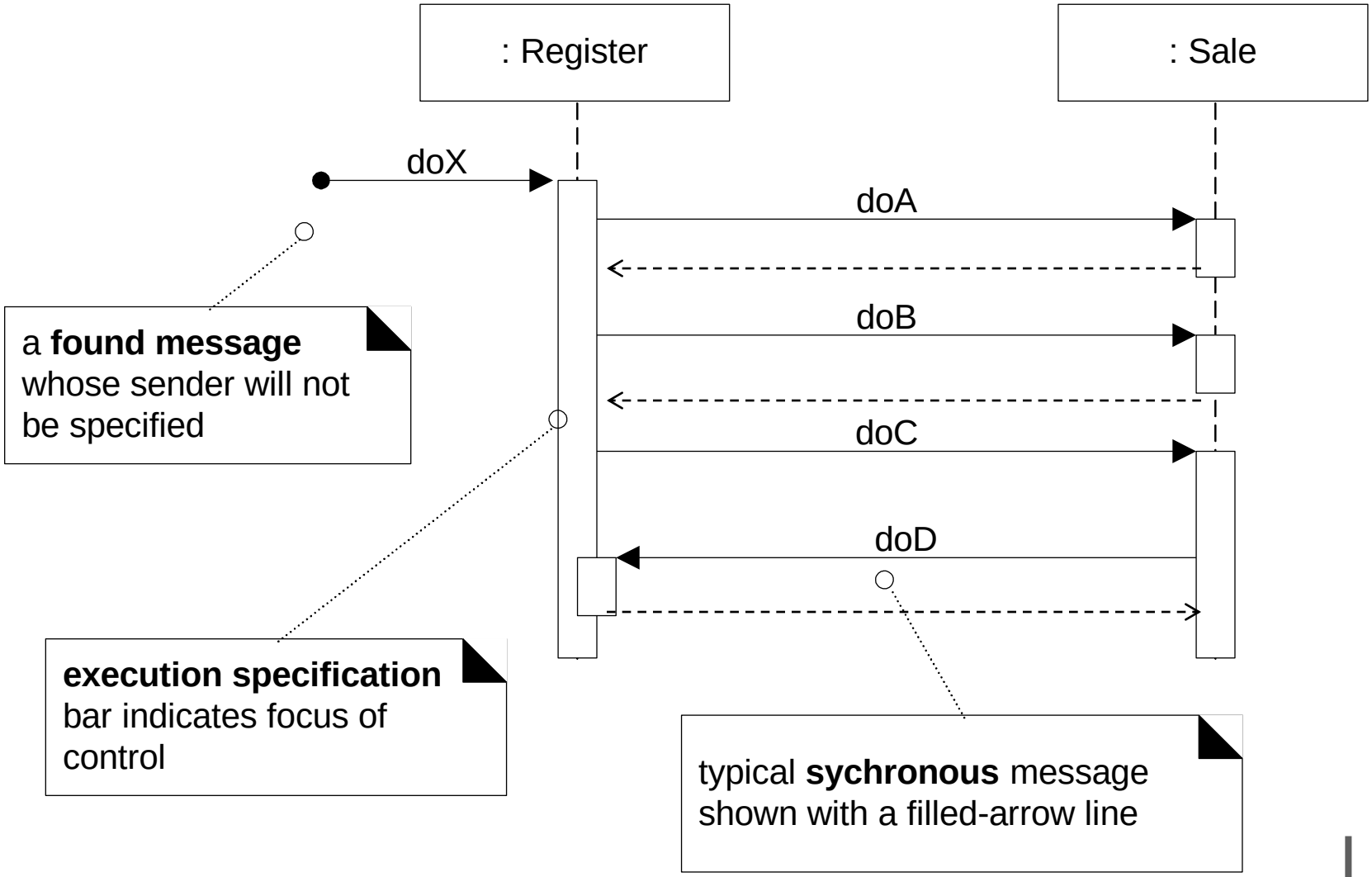
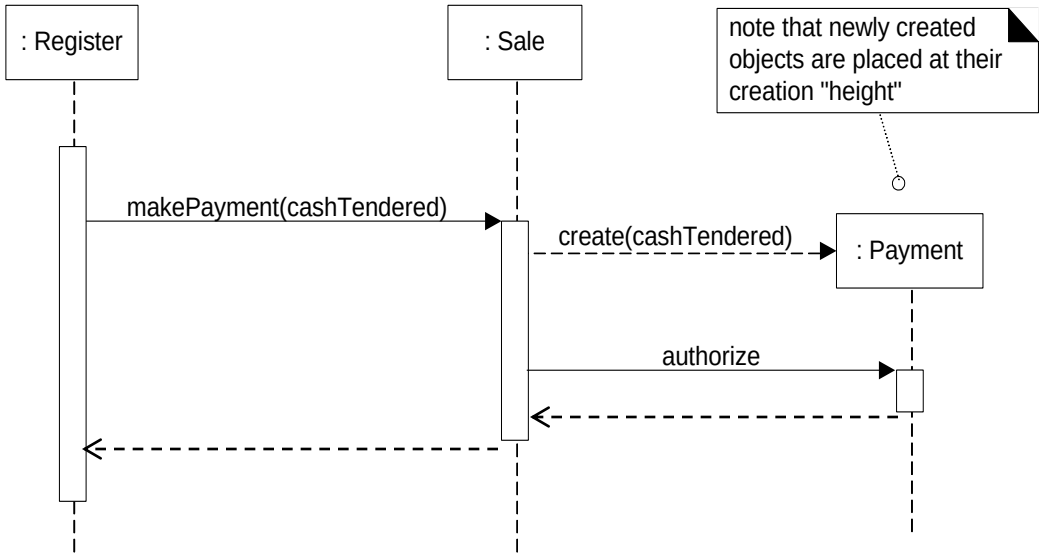
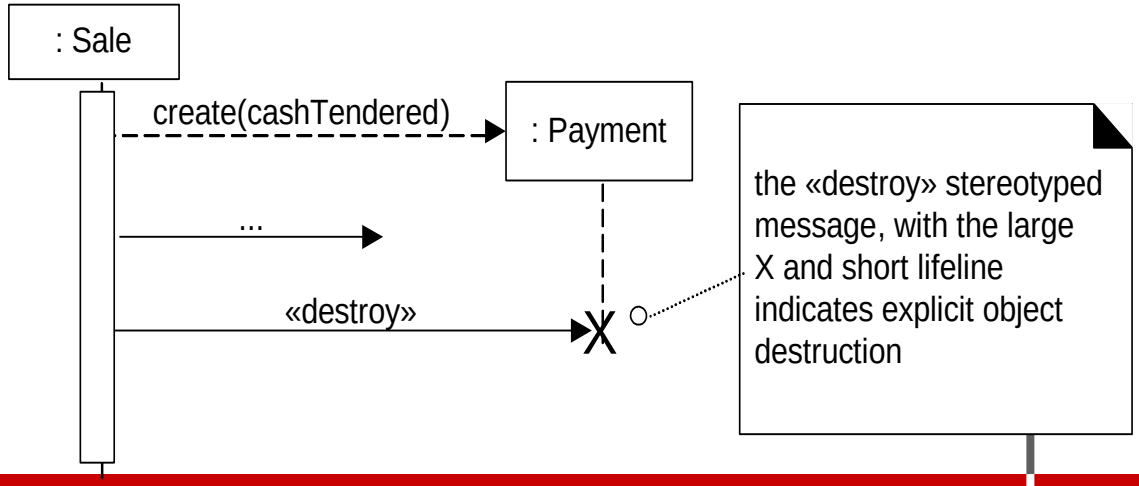


Diagramma di Sequenza





CombinedFragment

- La specifica di UML permette di esprimere comportamenti più complessi rispetto al singolo scambio di messaggi
- È possibile rappresentare l'esecuzione atomica di una serie di interazioni, oppure che un messaggio debba essere inviato solo in determinate condizioni
- A tale scopo UML mette a disposizione i *Combined Fragment*, cioè contenitori atti a delimitare un'area d'interesse nel diagramma
- Servono per spiegare che una certa catena di eventi, racchiusa in uno o più operandi, si verificherà in base alla semantica dell'operatore associato
- Ogni fragment ha un operatore e una (eventuale) guardia

CombinedFragment

- *Loop*: specifica che quello che è racchiuso nell'operando sarà eseguito ciclicamente finché la guardia sarà verificata
- *Alternatives (alt)*: indica che sarà eseguito il contenuto di uno solo degli operandi, quello la cui guardia risulta verificata
- *Optional (opt)*: indica che l'esecuzione del contenuto dell'operando sarà eseguita solo se la guardia è verificata
- *Break (break)*: ha la stessa semantica di opt, con la differenza che in seguito l'interazione sarà terminata
- *Critical*: specifica un blocco di esecuzione atomico (non interrompibile)
- *Parallel (par)*: specifica che il contenuto del primo operando può essere eseguito in parallelo a quello del secondo

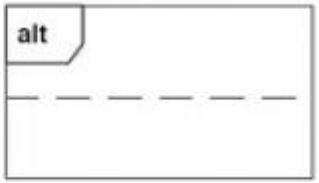
CombinedFragment

- *Weak Sequencing (seq)*: specifica che il risultato complessivo può essere una qualsiasi combinazione delle interazioni contenute negli operandi, purché:
  - l'ordinamento stabilito in ciascun operando sia mantenuto nel complesso
  - eventi che riguardano gli stessi destinatari devono rispettare anche l'ordine degli operandi, cioè i messaggi del primo operando hanno precedenza su quelli del secondo
  - eventi che riguardano destinatari differenti non hanno vincoli di precedenza vicendevole
- *Strict Sequencing (strict)*: indica che il contenuto deve essere eseguito nell'ordine in cui è specificato, anche rispetto agli operandi

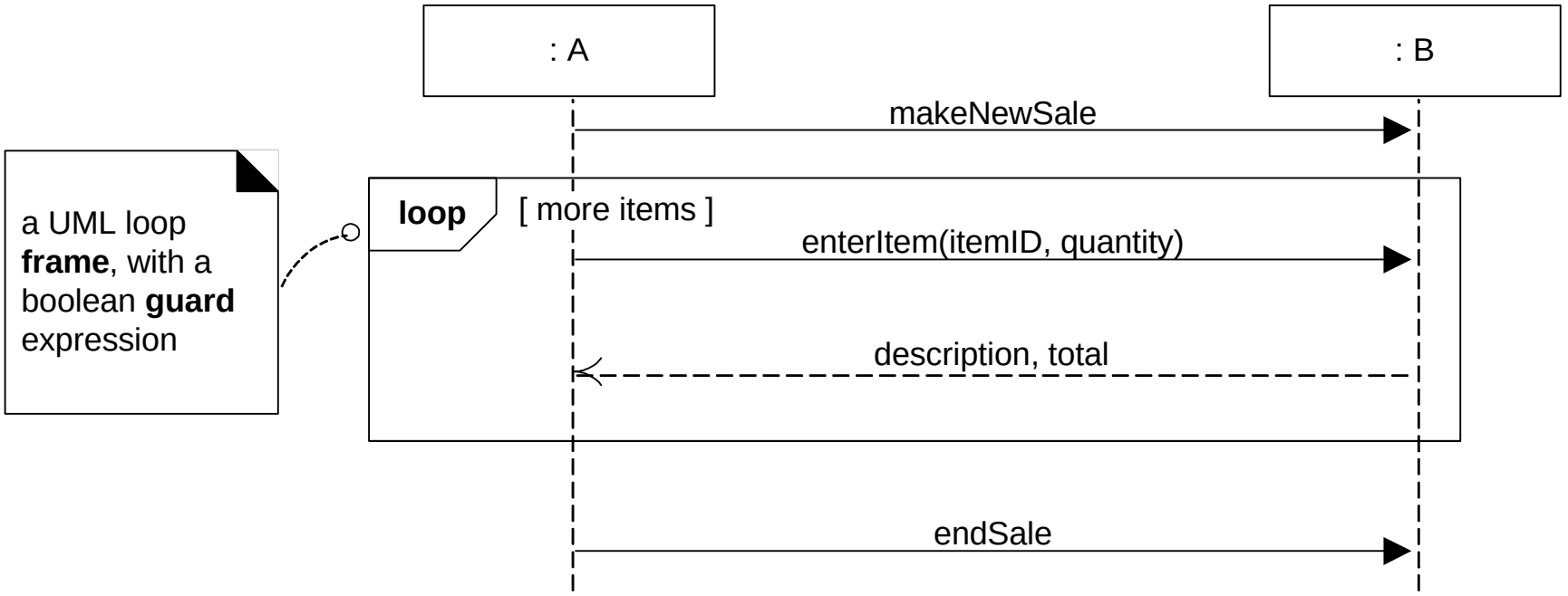
CombinedFragment

- *Ignore*: indica che alcuni messaggi, importanti ai fini del funzionamento del sistema, non sono stati rappresentati, perché non utili ai fini della comprensione dell'interazione
- *Consider*: è complementare ad ignore
- *Negative (neg)*: racchiude una sequenza di eventi che non deve mai verificarsi
- *Assertion (assert)*: racchiude quella che è considerata l'unica sequenza di eventi valida
  - di solito è associata all'utilizzo di uno State Invariant come rinforzo

CombinedFragment

Node Type	*Notation*	*Reference*
Frame		The notation shows a rectangular frame around the diagram with a name in a compartment in the upper left corner.
CombinedFragment		

CombinedFragment



CombinedFragment

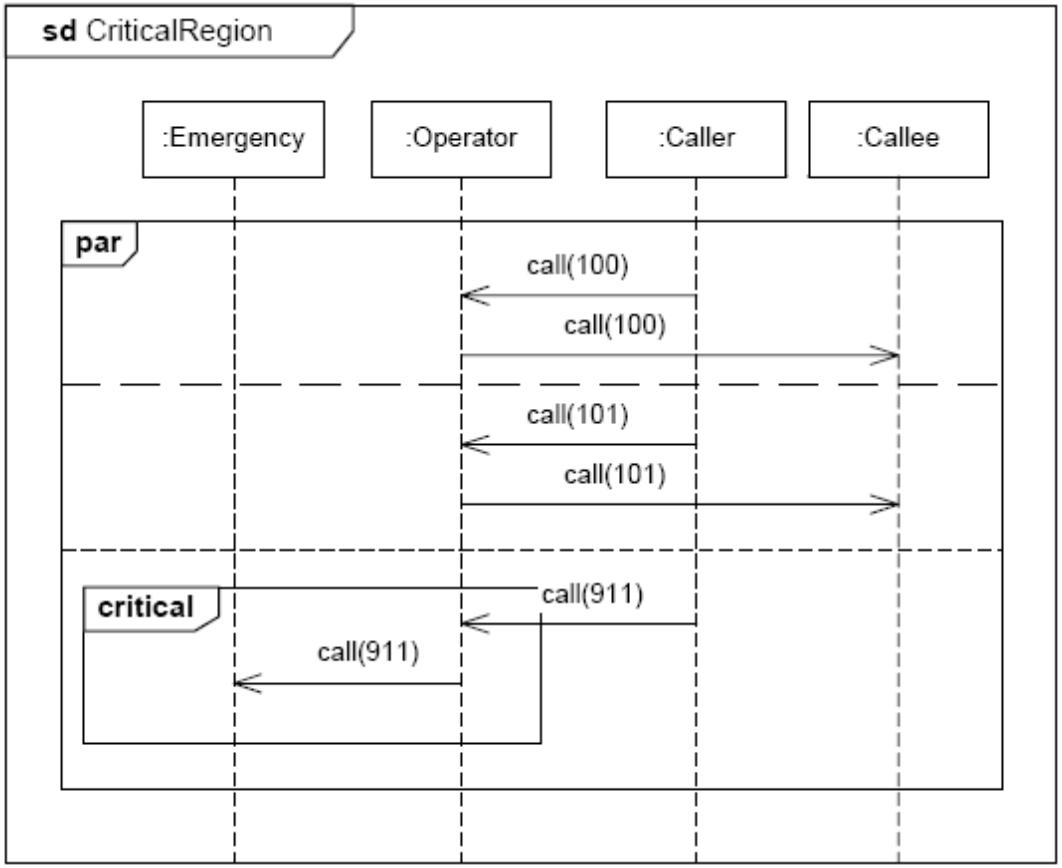


Diagramma di Stato

Diagramma di Stato

- I diagrammi di stato modellano la dipendenza
  che esiste tra lo stato di una classe / entità
  (cioè il valore corrente delle sue proprietà)
  e i messaggi e/o eventi che questo riceve in ingresso
- Specifica il ciclo di vita di una classe / entità,
  definendo le regole che lo governano
- Quando una classe / entità si trova in un certo stato
  può essere interessata a determinati eventi
  (e non ad altri)
- Come risultato di un evento una classe / entità
  può passare a un nuovo stato (transizione)

Diagramma di Stato

- Uno stato è una condizione o situazione nella vita di un oggetto in cui esso soddisfa una condizione, esegue un'attività o aspetta un evento
- Un evento è la specifica di un'occorrenza che ha una collocazione nel tempo e nello spazio
- Una transizione è il passaggio da uno stato a un altro in risposta ad un evento

Esempio

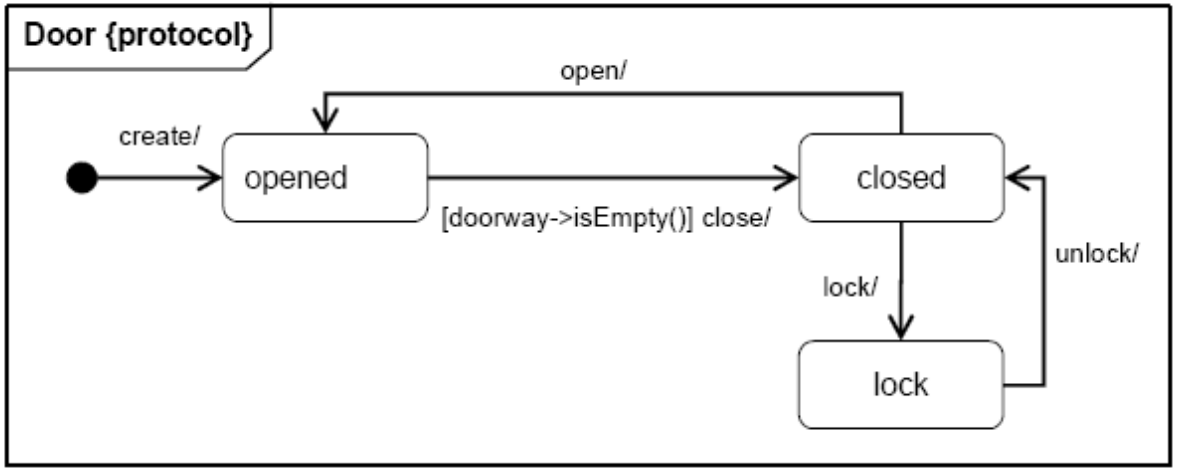
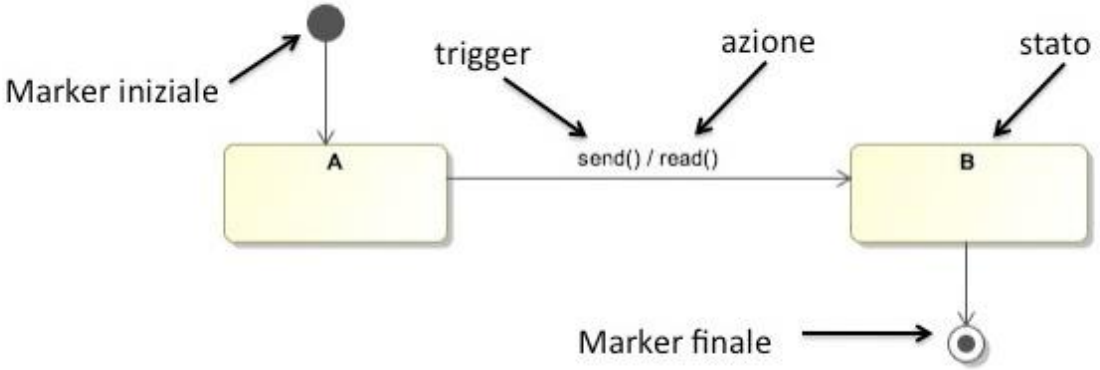

Door {protocol}
open/
create/
opened
closed
[doorway->isEmpty()] close/
unlock/
lock/
lock

Diagramma di Stato

- I concetti più importanti di un diagramma di stato sono:
  - gli *stati*, indicati con rettangoli con angoli arrotondati
  - le *transizioni* tra stati, indicate attraverso frecce
  - gli *eventi* che causano transizioni, la tipologia più comune è rappresentata dalla ricezione di un messaggio, che si indica semplicemente scrivendo il nome del messaggio con relativi argomenti vicino alla freccia
  - i *marker* di inizio e fine rappresentati rispettivamente da un cerchio nero con una freccia che punto allo stato iniziale e come un cerchio nero racchiuso da un anello sottile
  - le *azioni* che una entità è in grado di eseguire in risposta alla ricezione di un evento
  - il *vertice* che rappresenta l'astrazione di nodo nel diagramma e può essere la sorgente o la destinazione di una o più transizioni

Esempio


Marker iniziale
trigger
azione
stato
A
send() / read()
B
Marker finale

Stato

- Uno stato modella una situazione durante la quale vale una condizione invariante (solitamente implicita)
- L'invariante può rappresentare una situazione statica, come un oggetto in attesa che si verifichi un evento esterno
- Tuttavia, può anche modellare condizioni dinamiche, come il processo di esecuzione di un comportamento (cioè, l'elemento del modello in esame entra nello stato quando il comportamento inizia e lo lascia non appena il comportamento è completato)
- Stati possibili:
  - *Simple state*
  - *Composite state*
  - *Submachine state*

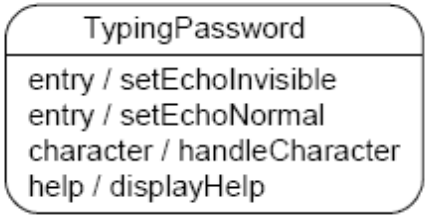
Stato

- *Composite state*:
  - può contenere una regione oppure è decomposto in due o più regioni ortogonali
  - ogni regione ha il suo insieme di sotto-vertici mutuamente esclusivi e disgiunti e il proprio insieme di transizioni
  - ogni stato appartenente ad una regione è chiamato substato
  - la transizione verso il marker finale all'interno di una regione rappresenta il completamento del comportamento di quella regione
  - quando tutte le regioni ortogonali hanno completato il loro comportamento questo rappresenta il completamento del comportamento dell'intero stato e fa scattare il trigger dell'evento di completamento

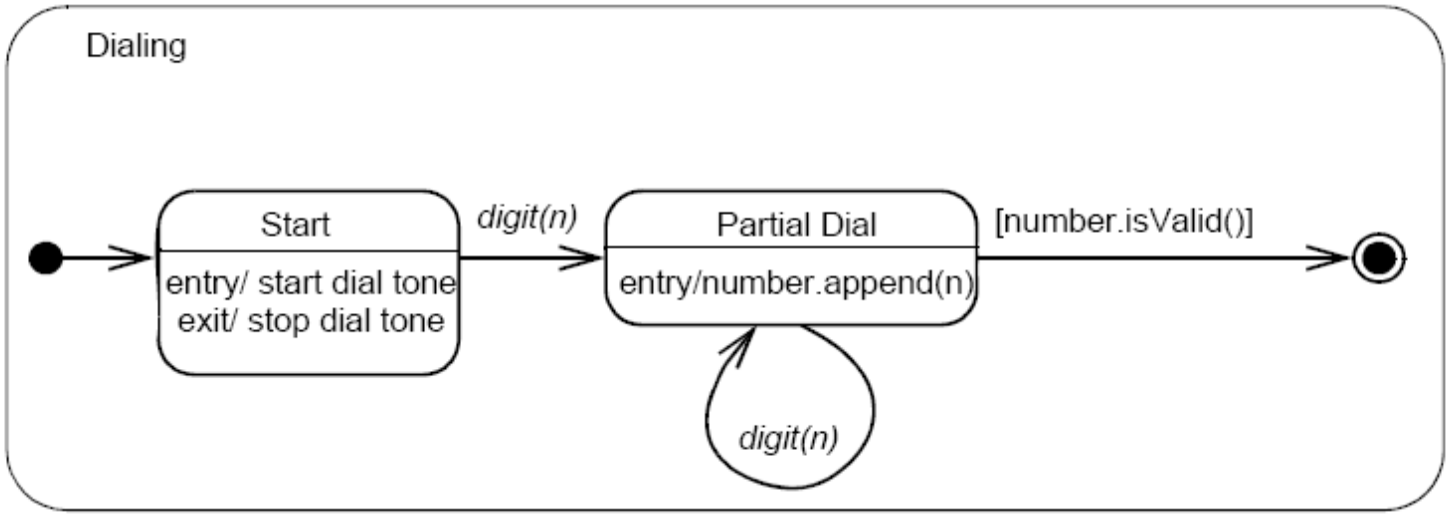
Stato

- *Simple state*:
  - è uno stato che non ha sottostati
- *Submachine state:*
  - specifica l'inserimento di un diagramma di una sotto-parte del sistema e permette la fattorizzazione di comportamenti comuni e il riuso dei medesimi
  - è semanticamente equivalente a un composite state, quindi le regioni del submachine state sono come le regioni del composite state
  - le azioni sono definite come parte dello stato
- Uno stato può essere ridefinito:
  - uno stato semplice può essere ridefinito diventando uno stato composito
  - uno stato composito può essere ridefinito aggiungendo regioni, vertici, stati, transizioni e azioni

Esempio



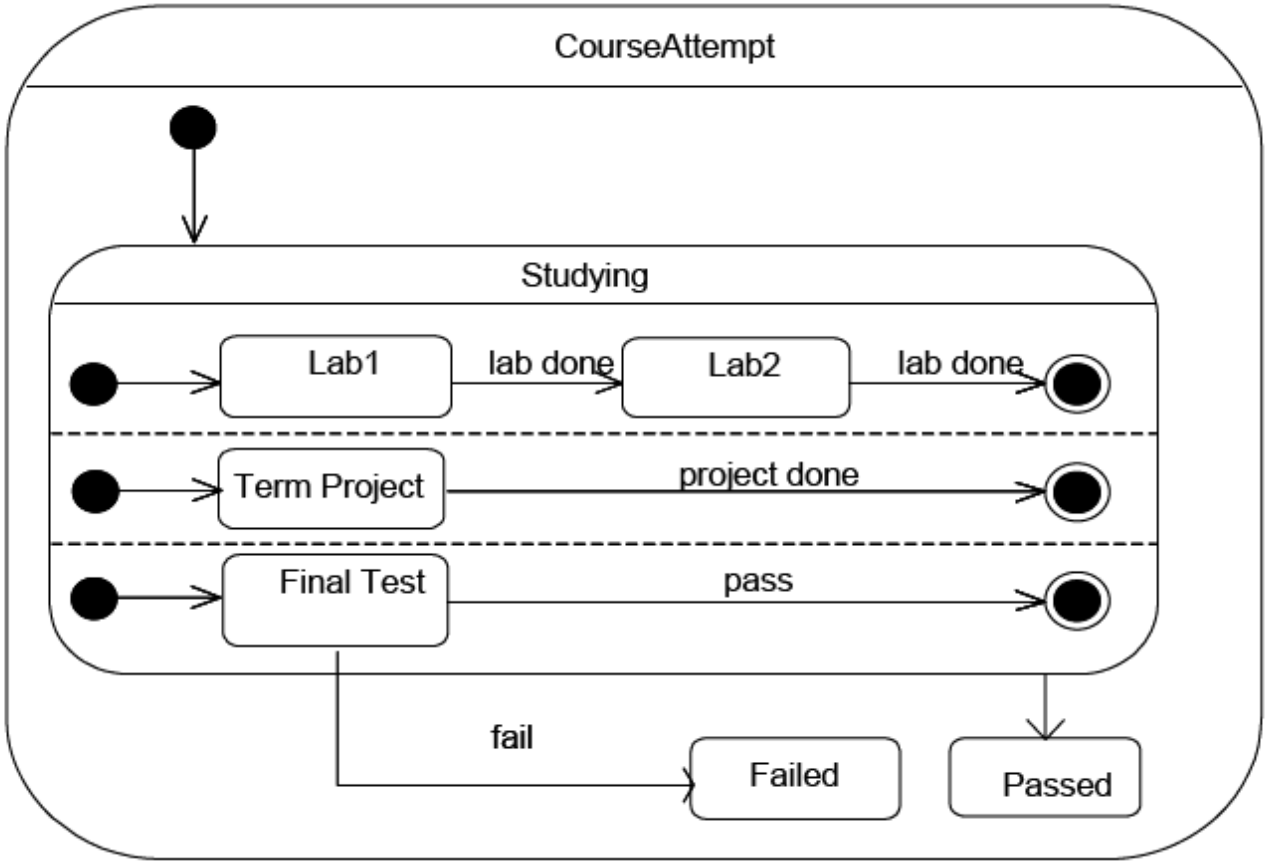
Simple state con azioni



Composite state

Esempio

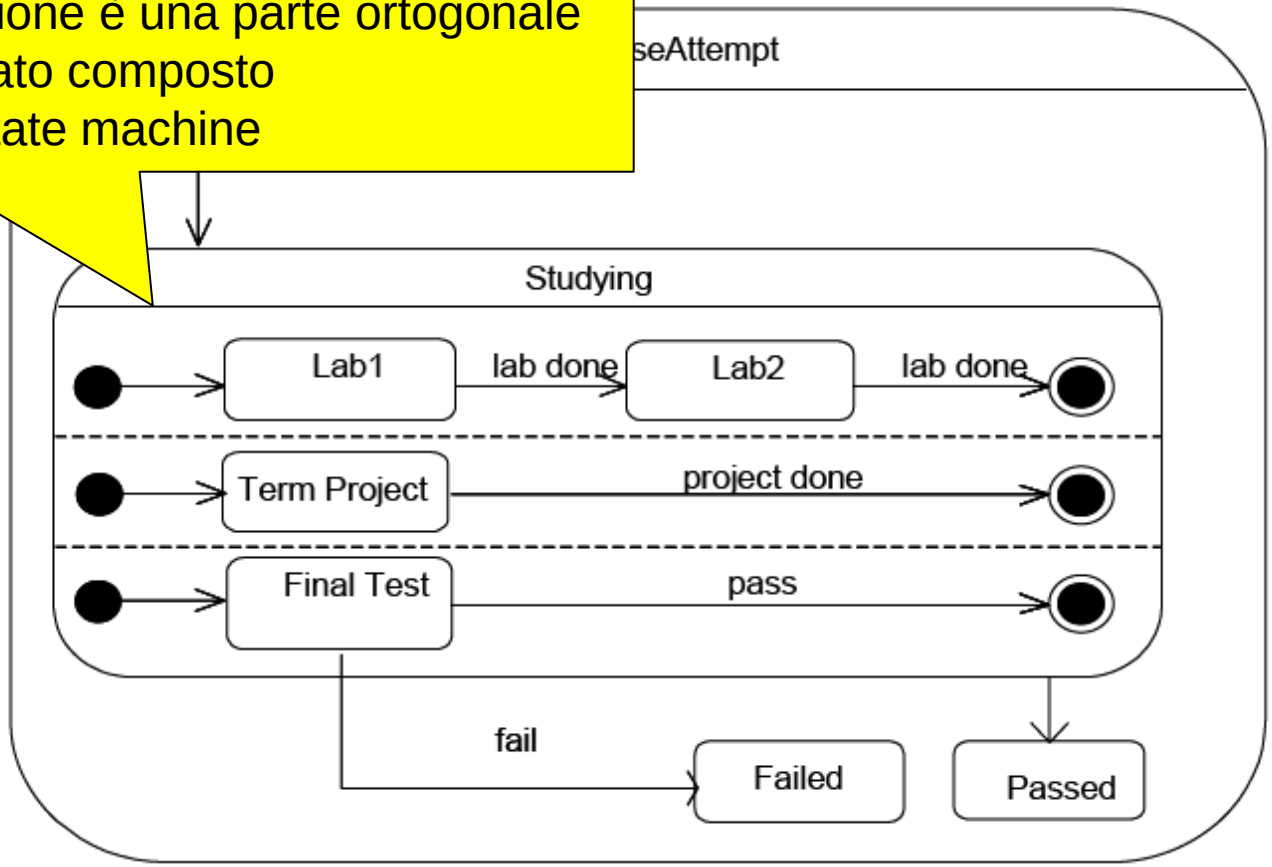
Composite state con regioni ortogonali



Esempio

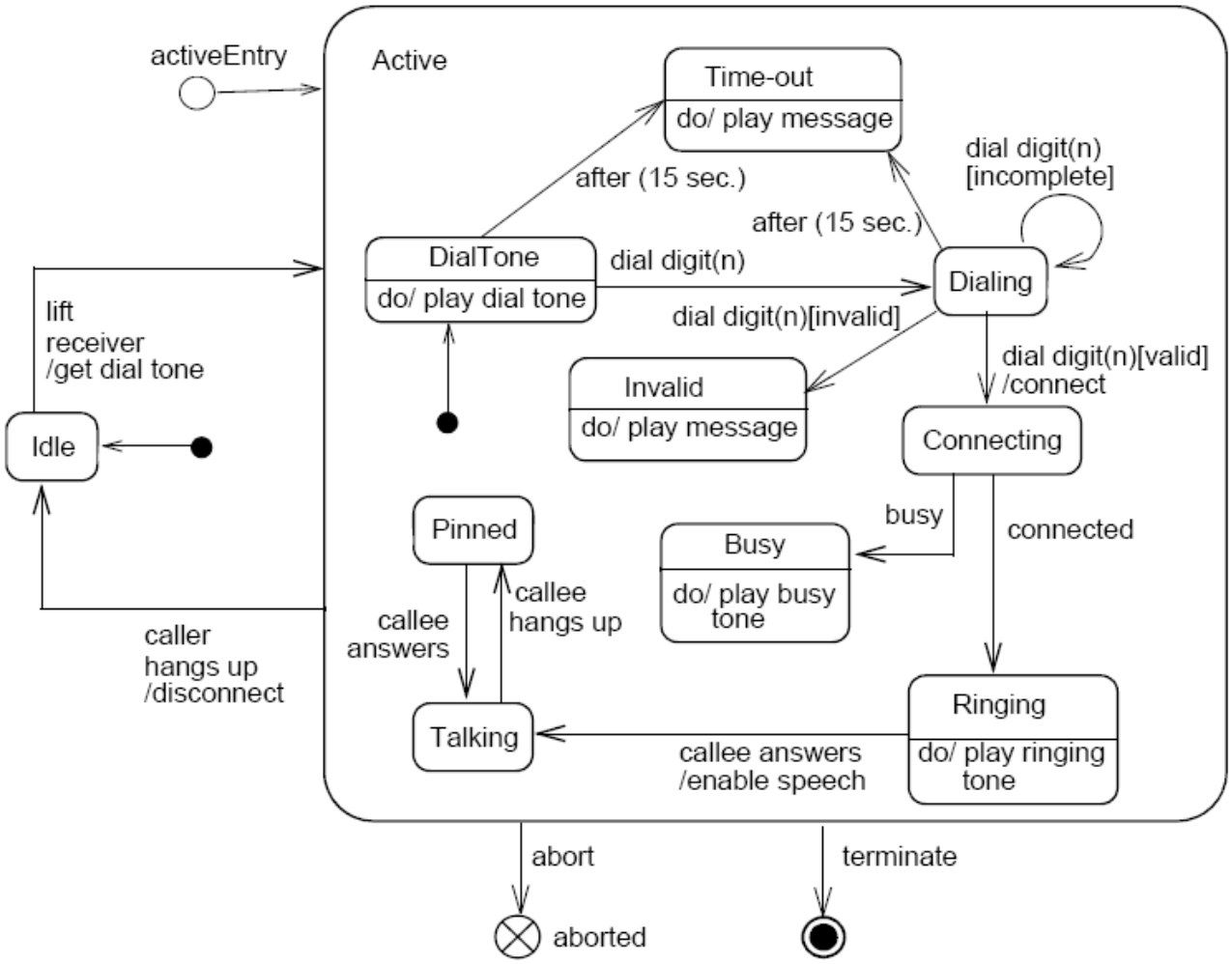
Composite state con regioni ortogonali

Una Regione è una parte ortogonale
di uno stato composto
o di un state machine



Esempio

Submachine state



Pseudostato

- Gli pseudostati vengono generalmente utilizzati per collegare più transizioni in percorsi di transizioni di stato più complessi
- Ad esempio, combinando una transizione che entra in uno pseudostato fork con un insieme di transizioni che escono dallo pseudostato fork, otteniamo una transizione composta che porta a un insieme di stati ortogonali

Tipi di Pseudostati

- *Initial*: vertice di default che è la sorgente della singola transizione verso lo stato di default di un composite state
  - Ci può essere al massimo un vertice initial e la transizione uscente da questo vertice non può avere trigger o guardie
- *deepHistory*: la più recente configurazione attiva del composite state che contiene direttamente questo pseudostato
- *shallowHistory:* il più recente substate attivo di un composite state

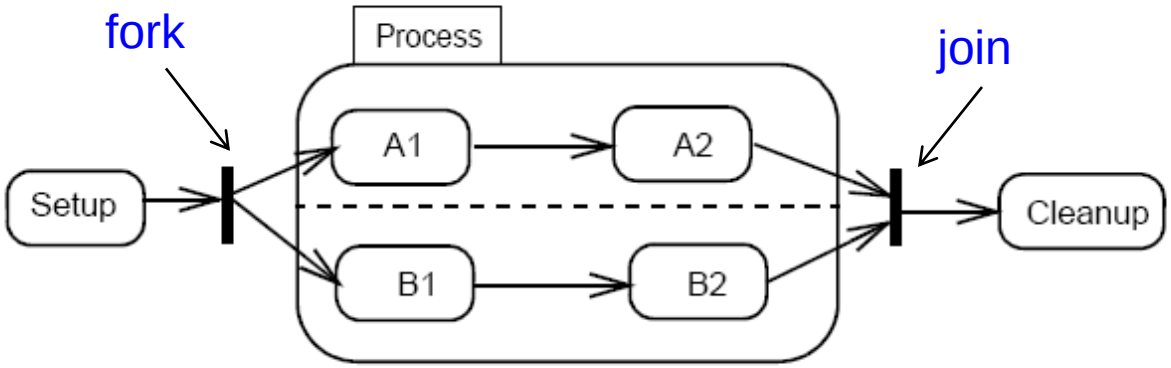
initial

deepHistory

shallowHistory

Tipi di Pseudostati

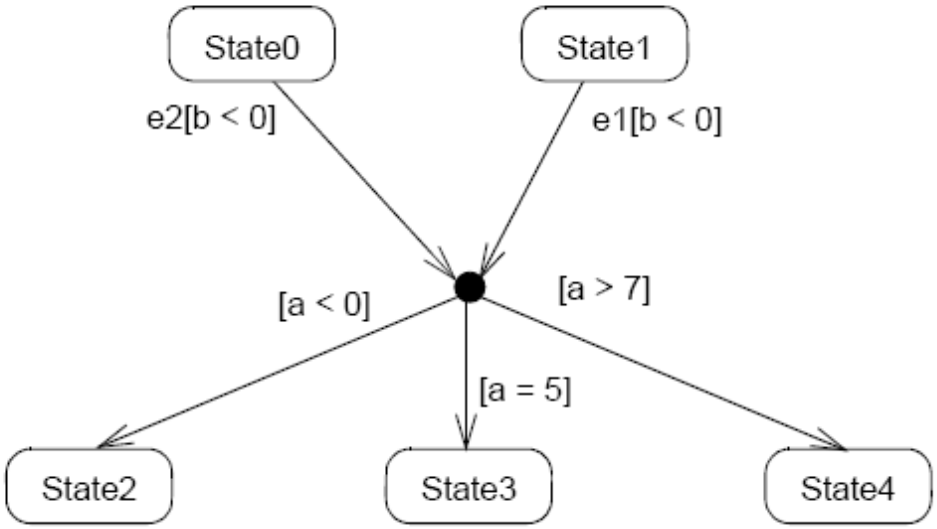
- *Join:* permette di eseguire il merge di diverse transizioni provenienti da diverse sorgenti appartenenti a differenti regioni ortogonali
  - Le transizioni entranti in questo vertice non possono avere guardie e trigger
- *Fork:* permette di separare una transizione entrante in due o più transizioni che terminano in vertici di regioni ortogonali
  - Le transizioni uscenti da questo vertice non possono avere guardie



Tipi di Pseudostati

- *Junction:* è un vertice privo di semantica che viene usato per “incatenare” insieme transizioni multiple
  - È usato per costruire percorsi di transizione composti tra stati
  - Una junction può essere usata per far convergere transizioni multiple entranti in una singola transizione uscente che rappresenta un percorso condiviso di transizione
  - Allo stesso modo una junction può essere usata anche per suddividere una transizione entrante in più transizioni uscenti con differenti guardie
  - Permette di realizzare un branch condizionale statico
  - Esiste una guardia predefinita chiamata “else” che viene attivata quando tutte le guardie sono false

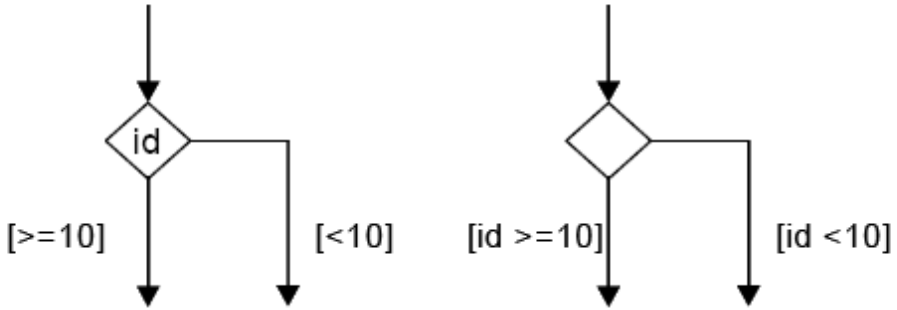
Junction



Tipi di Pseudostati

- *Choice:* è un tipo di vertice che quando viene raggiunto causa la valutazione dinamica delle guardie dei trigger delle transizioni uscenti
  - Le guardie sono quindi tipicamente scritte sotto forma di “funzione” che viene valutata al momento del raggiungimento del vertice choice
  - Permette di realizzare un branch condizionale dinamico separando le transizioni uscenti e creando diversi percorsi di uscita
  - Se più di una guardia è verificata viene scelto il percorso in modo arbitrario
  - Se nessuna è verificata il modello viene considerato “ill-formed”
  - È quindi caldamente consigliato definire sempre un percorso di uscita di tipo “else”

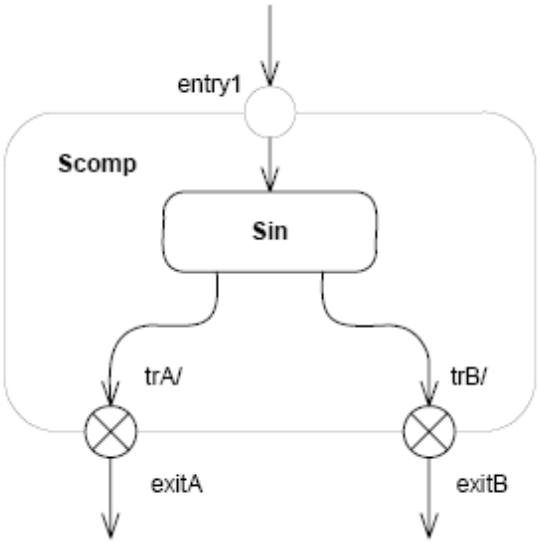
Choice



Tipi di Pseudostati

- *Entry point:* è l'ingresso di uno state machine
  o di un composite state
- *Exit point:* è l'uscita da uno state machine
  o da uno stato composito
  - Entrare in questo vertice significa attivare la transizione che ha questo vertice come sorgente
- *Terminate:* entrare in questo vertice implica
  che l'esecuzione di questo state machine è terminata
  - Lo state machine non uscirà da nessuno stato
    né verranno invocate altre azioni a parte quelle associate con la transizione che porta allo stato terminate

Tipi di Pseudostati



Terminate pseudostate

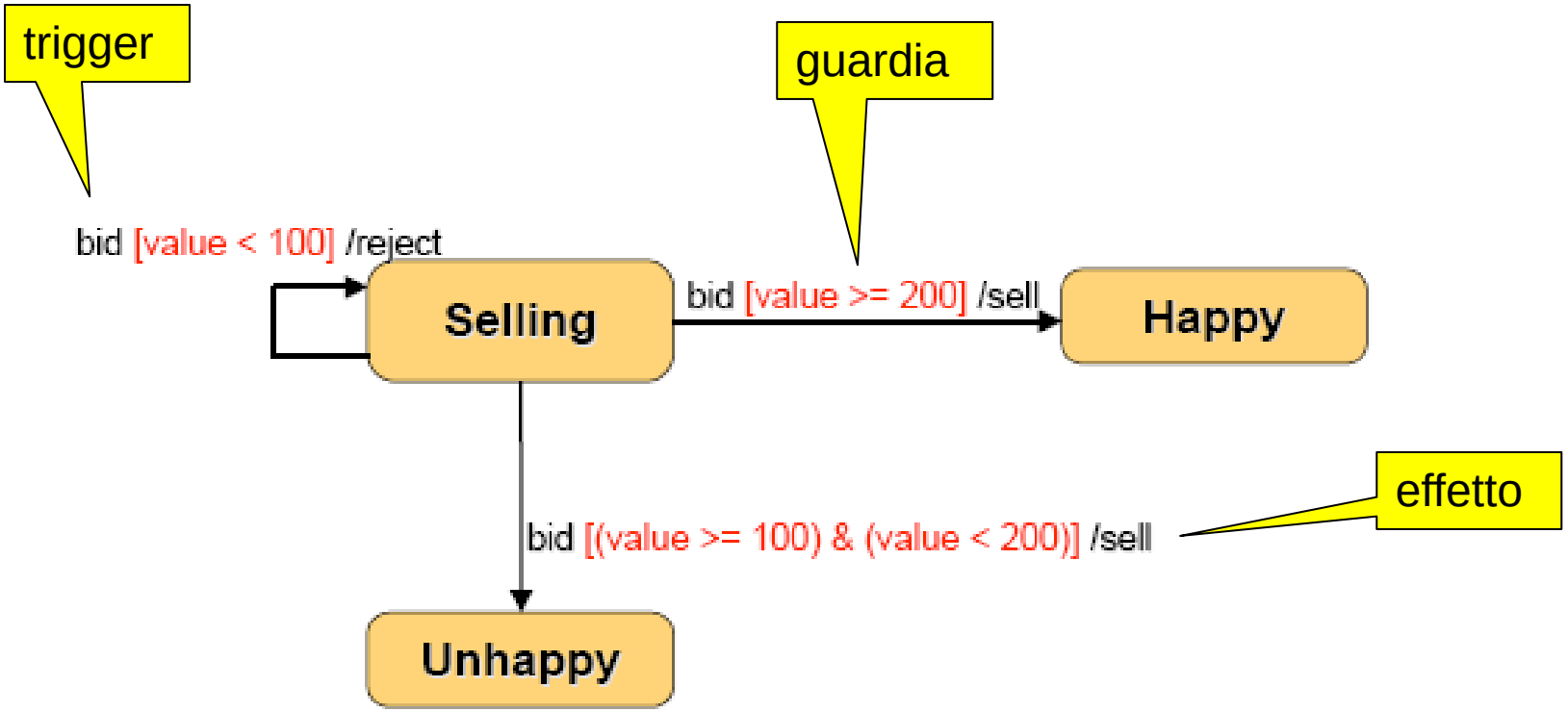
Transizione

- Una transizione è una relazione diretta tra un vertice di origine e un vertice di destinazione
- Può essere parte di una transizione composta, che porta la macchina a stati da una configurazione di stato a un’altra, rappresentando la risposta completa della macchina a stati al verificarsi di un evento di un tipo particolare

Transizione

- Analizzando la specifica UML è possibile vedere che tra le proprietà di una transizione compaiono diversi concetti molto rilevanti tra cui:
  - *trigger*: cioè i tipi di evento che possono innescare la transizione
  - *guardia*: cioè un vincolo che fornisce un controllo fine sull'innesco della transizione
    - La guardia è valutata quando una occorrenza dell'evento è consegnata allo stato
    - Se la guardia risulta verificata allora la transizione può essere abilitata altrimenti questa viene disabilitata
    - Le guardie dovrebbero essere pure espressioni senza side effect, i quali sono assolutamente sconsigliati nella specifica
  - *effetto*: specifica un comportamento opzionale (cioè un'azione) che deve essere eseguito quando la transizione scatta

Transizione



Tipi di Eventi

- *Evento di chiamata*: ricezione di un messaggio che richiede l'esecuzione di una operazione
- *Evento di cambiamento*: si verifica quando una condizione passa da "falsa" a "vera"
  - Tale evento si specifica scrivendo <<when>> seguito da un'espressione, racchiusa tra parentesi tonde, che descrive la condizione
  - Utile per descrivere la situazione in cui un oggetto cambia stato perché il valore dei suoi attributi è modificato dalla risposta a un messaggio inviato

Tipi di Eventi

- *Evento segnale*: consiste nella ricezione di un segnale
- *Evento temporale*: espressione che denota un lasso di tempo che deve trascorrere dopo un evento dotato di nome
  - Con <<after>> si può far riferimento all'istante in cui l'oggetto è entrato nello stato corrente
  - Con <<at>> si esprime qualcosa che deve accadere in un particolare momento

Azione

- Un’azione è un elemento avente un nome
  che è l’unità fondamentale della funzionalità eseguibile
- L’esecuzione di un’azione rappresenta
  una trasformazione o elaborazione
  nel sistema modellato, sia esso un sistema informatico
  o altro

Azione

- *Entry*: tutte le volte che si entra in uno stato
  viene generato un evento di entrata
  a cui può essere associato uno o più specifici
  comportamenti che vengono eseguiti
  prima che qualsiasi altra azione possa essere eseguita
- *Exit*: tutte le volte che si esce da uno stato
  viene generato un evento di uscita
  a cui può essere associato uno o più specifici
  comportamenti che vengono eseguiti
  come ultimo passo prima che lo stato venga lasciato

Azione

- *Do*: rappresenta il comportamento che viene eseguito all'interno dello stato
  - il comportamento parte subito dopo l'ingresso nello stato (dopo che è stato eseguito l'entry)
  - se questo comportamento termina mentre lo stato è ancora attivo viene generato un evento di completamento e nel caso sia stata specificata una transizione per il completamento allora viene eseguito l'exit e successivamente la transizione d'uscita
  - se invece viene innescata una transizione di uscita prima che il do termini, allora l'esecuzione del do è abortita, viene eseguito l'exit e poi la transizione

Azione

- Il comportamento di entry è stato definito
  per permettere di fattorizzare comportamenti comuni
  all'ingresso di uno stato
- Senza la possibilità di poter specificare
  un comportamento di entry, il progettista avrebbe dovuto
  specificare su ogni transizione verso lo stato
  una azione effetto legata alle transizioni
- Il medesimo discorso vale per le exit,
  nella quale sono fattorizzati tutti quei comportamenti
  che devono essere eseguiti prima di uscire dallo stato
- È importante però ricordare che non *vi è la possibilità*
  di esprimere condizioni di *guardia* su questi
  comportamenti

Diagramma di Stato

- Transizione interna che esclude azioni di entrata e di uscita

- Attività interna: generazione di un thread concorrente che resta in esecuzione finché:
  - il thread (eventualmente) termina
  - si esce dallo stato attraverso una transizione esterna

Diagramma di Stato

- Il diagramma di stato relativo a una singola classe / entità dovrebbe essere il più semplice possibile
- Le classi / entità con diagrammi di stato complicati hanno diversi problemi:
  - il codice risulta più difficile da scrivere perché le implementazioni dei metodi contengono molti blocchi condizionali
  - il testing è più arduo
  - è molto difficile utilizzare una classe dall'esterno se il comportamento dipende dallo stato in modo complesso

Diagramma di Stato

- Se una classe ha un numero troppo elevato di stati
  è bene considerare se esistano progetti alternativi,
  come per esempio scomporre la classe
  in due diverse classi
- Questa però non va presa come regola universale
- La scomposizione di una classe è sempre un'operazione
  molto delicata e va ponderata molto bene
  in quanto potrebbe portare a progetti fuorvianti o instabili
- A volte il male minore è proprio avere diagrammi di stato
  complessi

Diagramma delle Attività

Diagramma delle Attività

- I diagrammi delle attività descrivono il modo in cui diverse attività sono coordinate e possono essere usati per mostrare l'implementazione di una operazione
- Un diagramma delle attività mostra le attività di un sistema in generale e delle sotto-parti, specialmente quando un sistema ha diversi obiettivi e si desidera modellare le dipendenze tra essi prima di decidere l'ordine in cui svolgere le azioni
- I diagrammi delle attività sono utili anche per descrivere lo svolgimento dei singoli casi d'uso e la loro eventuale dipendenza da altri casi

Diagramma delle Attività

- Sostanzialmente, modellano un processo
- Organizzano più entità in un insieme di azioni secondo un determinato flusso
- Si usano ad esempio per:
  - modellare il flusso di un caso d'uso (analisi)
  - modellare il funzionamento di un'operazione (progettazione)
  - modellare un algoritmo (progettazione)

Diagramma delle Attività

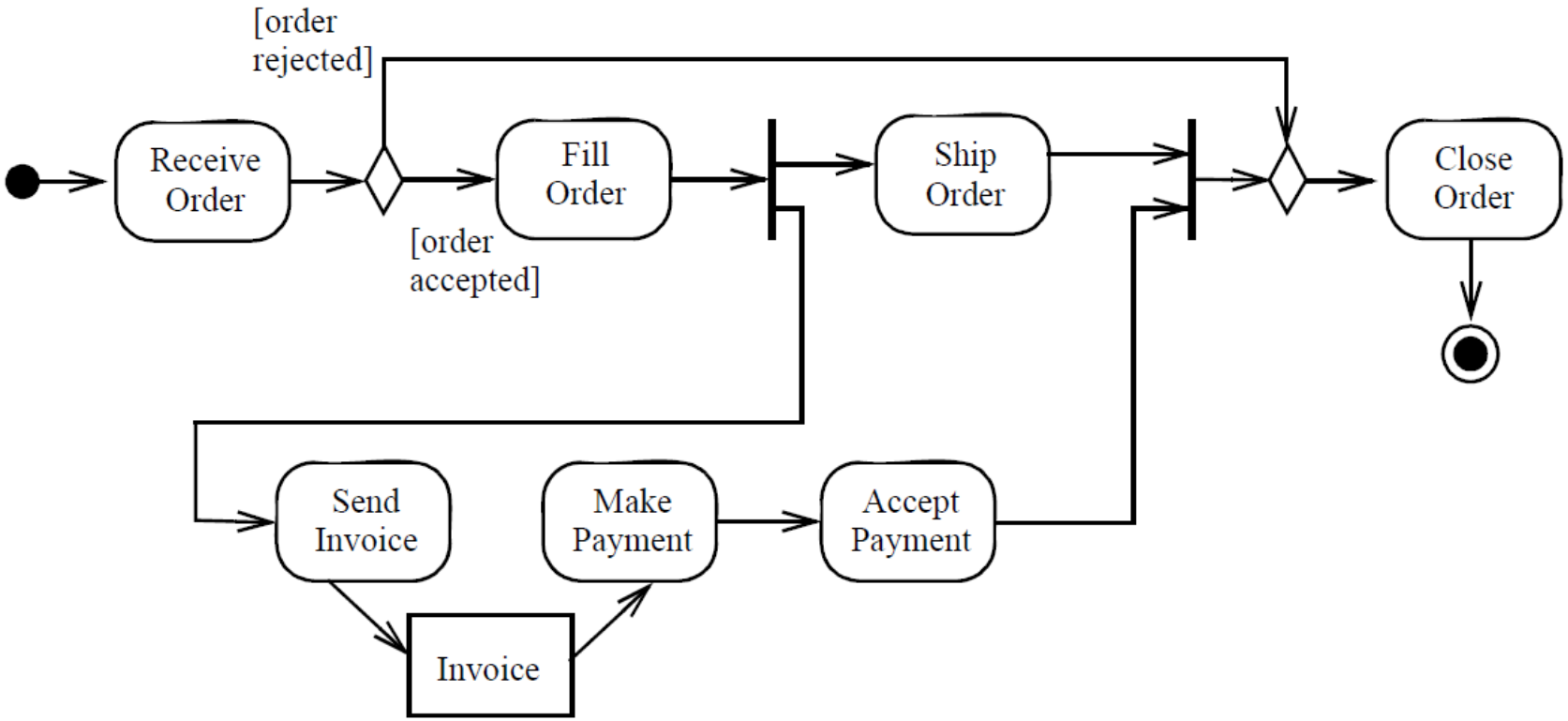


Diagramma delle Attività

- *Attività*: indica un lavoro che deve essere svolto
  - Specifica un'unità atomica, non interrompibile e istantanea di comportamento
  - Da ogni attività possono uscire uno o più **archi**, che indicano il percorso da una attività ad un'altra
- *Arco*: è rappresentato come una freccia proprio come una transizione
  - A differenza di una transizione, un arco non può essere etichettato con eventi o azioni
  - Un arco può essere etichettato con una condizione di guardia, se l'attività successiva la richiede

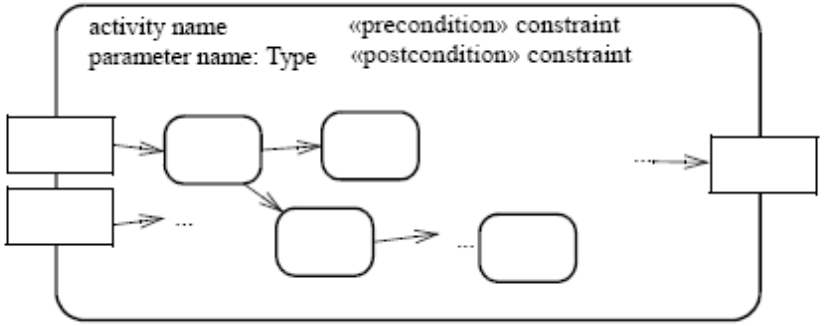
Diagramma delle Attività

- *Oggetto*: rappresenta un oggetto “importante” usato come input/output di azioni
- *Sottoattività*: “nasconde” un diagramma delle attività interno a un’attività
- *Start e End Point*: punti di inizio e fine del diagramma
  - Gli End Point possono anche non essere presenti, oppure essere più di uno
- *Controllo*: nodi che descrivono il flusso delle attività

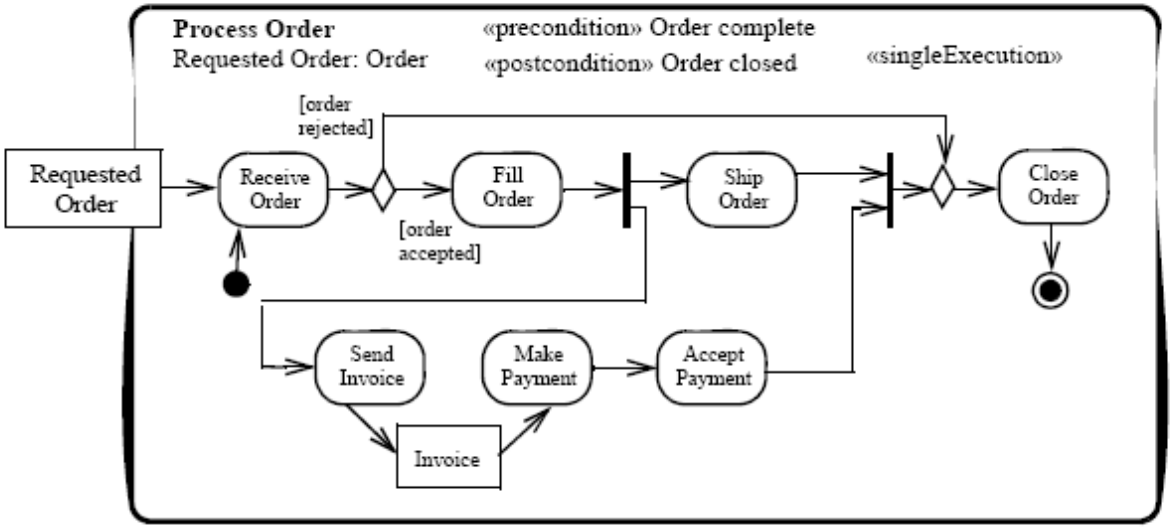
Diagramma delle Attività

- Per capire la semantica dei diagrammi di attività, bisogna immaginare delle entità, dette *token*, che viaggiano lungo il diagramma
- Il flusso dei token definisce il flusso dell'attività
- I token possono rimanere fermi in un nodo azione/oggetto in attesa che si avveri
  - una condizione su un arco
  - oppure una precondizione o postcondizione su un nodo
- Il movimento di un token è atomico
- Un nodo azione viene eseguito quando sono presenti token su tutti gli archi in entrata, e tutte le precondizioni sono soddisfatte
- Al termine di un'azione, sono generati token su tutti gli archi in uscita

Notazione



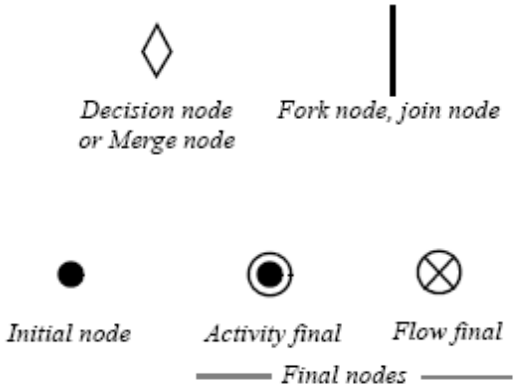
Attività



Attività con parametro di ingresso

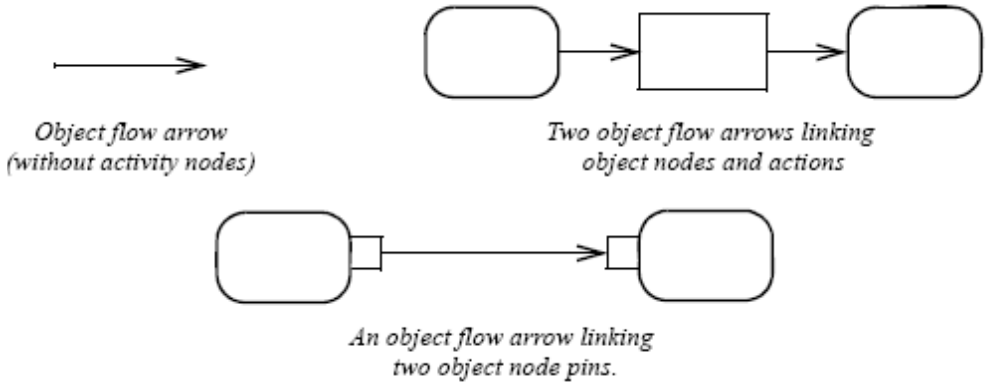
Diagramma delle Attività

- *Decision e merge* : determinano il comportamento condizionale:
  - decision ha una singola transizione entrante
    e più transizioni uscenti in cui solo una di queste sarà prescelta
  - merge ha più transizioni entranti e una sola uscente e serve
    a terminare il blocco condizionale cominciato con un decision
- *Fork e join*: determinano il comportamento parallelo:
  - quando scatta la transazione entrante, si eseguono in parallelo
    tutte le transazioni che escono dal fork
    - Con il parallelismo non è specificata la sequenza
    - Le fork possono avere delle guardie
  - Per la sincronizzazione delle attività
    è presente il costrutto di join che ha
    più transazioni entranti e una sola
    transazione uscente

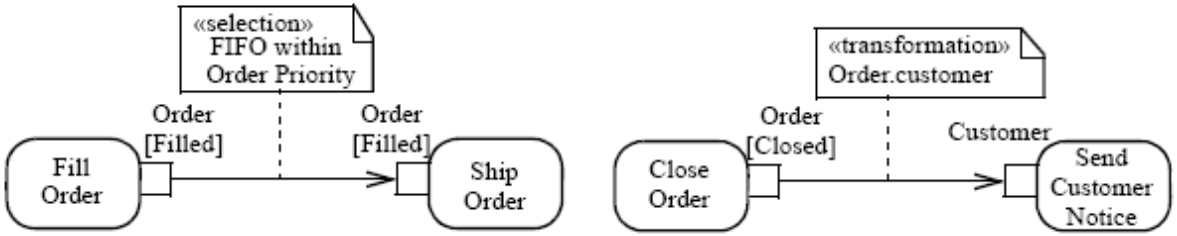


Object Flow

- È un arco che ha oggetti o dati che fluiscono su di esso



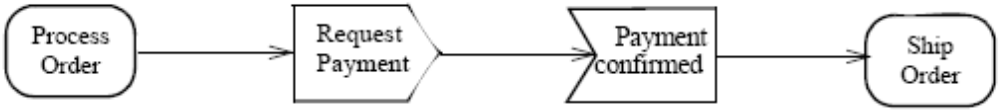
Notazione



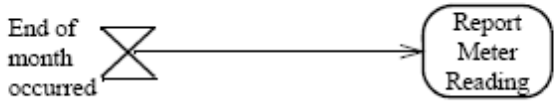
Esempio

Segnali ed Eventi

SendSignal Action

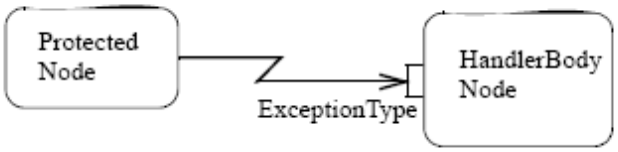


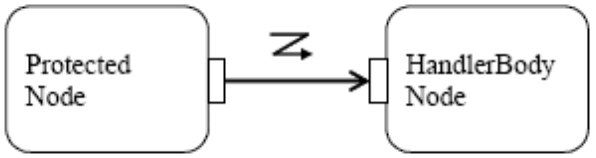
AcceptSignal Action

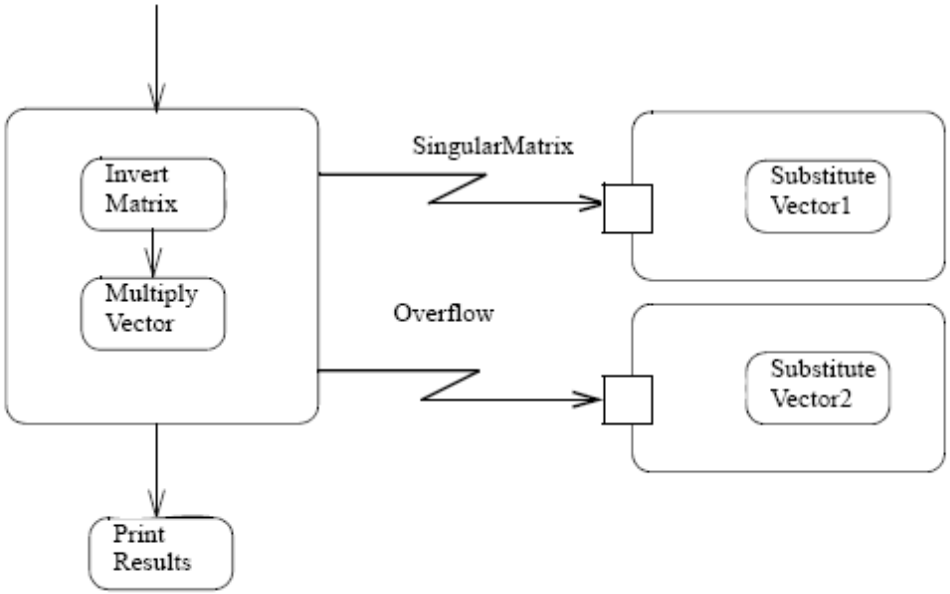


Evento ripetuto nel tempo

Exception Handler







InterruptibleActivityRegion